PIERRE LEBEUX

INTRODUCTION AU BASIC

SYBEX

SYBEX

INTRODUCTION AU BASIC

SUR

MICRO-ORDINATEURS

P. LE BEUX

Ing. E.C.P.
PhD Univ. Californie
Dr es Sciences



Imprimé en France.

ISBN : 2-902414-16-1

TABLE DES MATIERES

AVANT-PROPOS

Cet ouvrage s'adresse à la fois au lecteur débutant en informatique et au lecteur ayant déjà eu l'expérience des langages de programmation.

Pour la première catégorie de lecteurs, il n'y a aucune connaissance préalable nécessaire, et nous conseillons donc au lecteur de suivre le livre pas à pas pour acquérir les concepts de base nécessaires à la compréhension des chapitres suivants. Pour la deuxième catégorie de lecteurs, l'approche du livre doit être différente ; ils pourront se reporter directement au chapitre II pour connaître les éléments de base du langage et, dans la suite, s'intéresser aux particularités du langage et essayer de programmer directement en BASIC les exercices qui sont proposés.

Les derniers chapitres sont plus spécialisés et présenteront un intérêt pour tous ceux qui veulent connaître les extensions non standard du langage.

En résumé, le chapitre I est une introduction au vocabulaire et aux concepts de base de l'informatique.

Le chapitre II présente des généralités sur le langage BASIC.

Le chapitre III décrit les instructions fondamentales du langage avec des exercices associés.

Le chapitre IV présente les concepts plus élaborés du langage de base, notamment le traitement de listes, de tableaux, de matrices, la notion de fonction et de sous-programmes.

Dans le chapitre V sont abordés des problèmes de traitement de fichiers qui, bien qu'étant particuliers à chaque système, sont présentés de façon suffisamment générale pour permettre de s'adapter à n'importe quel système.

Dans le chapitre VI sont présentés les traitements graphiques et les possibilités offertes pour que certains systèmes soient étudiés sur des programmes types qu'il est facile d'adapter sur n'importe quel système.

Enfin le chapitre VII renferme un certain nombre de facilités offertes par les systèmes BASIC, notamment sur micro-ordinateurs.

Nous espérons ainsi que le lecteur de cet ouvrage trouvera réponse à toutes les questions qu'il peut se poser concernant le langage BASIC ou ses extensions.

PREFACE

BASIC est un langage de programmation de haut niveau. Il s'agit d'un acronyme pour « Beginner's all purpose symbolic instruction code » et il a été développé initialement au « Darmouth College ».

C'est un langage interactif et interprété, c'est-à-dire que chaque instruction est traduite en langage machine et exécutée immédiatement lorsqu'elle est reconnue comme étant correcte. S'il y a des erreurs, elles sont ainsi détectées tout de suite et l'on peut les corriger. De ce fait, BASIC est un langage qui est facile à apprendre et à utiliser pour des débutants. Son caractère interactif facilite l'accès à l'ordinateur sans qu'il soit nécessaire de se familiariser avec les idiosyncrasies des systèmes d'exploitation des ordinateurs : cartes contrôles au langage abscons pour les débutants, messages d'erreurs bien souvent peu explicites, délais d'attente...

Il faut noter cependant que ce langage peut également être utilisé par des professionnels qui veulent tester rapidement un algorithme, traiter un problème de mathématique ou de statistique élémentaire ou même consulter une petite base de donnée personnelle. BASIC est le langage qui a le plus souvent été utilisé sur des machines travaillant en temps partagé. Le développement de la technologie des micro-ordinateurs et des systèmes personnels a redonné au langage BASIC un regain d'intérêt qui est dû essentiellement à sa facilité d'apprentissage et à son caractère interactif. C'est pour cette raison que nous pensons qu'il est utile d'écrire un ouvrage de base sur le langage et, en particulier, de présenter les versions actuelles, disponibles sur les différents types de micro-ordinateurs.

Il n'en reste pas moins qu'il existe de nombreux détracteurs du langage : il est ancien et ne permet pas de bien structurer les programmes, son caractère interactif peut donner des habitudes de programmation détestables. D'autre part, les programmes écrits en BASIC s'exécutent plus lentement que dans d'autres langages, et c'est un langage à base d'anglais, ce qui le rend moins adapté à l'enseignement dans des pays non anglo-saxons... Pour ce qui est

des premières critiques, elles sont dans une large mesure justifiées : il existe des langages plus modernes, mais ils ne sont pas interprétés (si l'on exclut APL). Le caractère interactif peut donner des programmes mal construits, mais que l'on peut considérer comme des « brouillons » que l'on jette et que l'on peut éventuellement utiliser comme base pour les reprogrammer de manière structurée dans un autre langage.

En ce qui concerne les dernières critiques elles ne sont que partiellement valides. La vitesse d'exécution n'a plus de sens lorsque l'utilisateur obtient une réponse quasi instantanée à son niveau. En effet, dans bien des cas, la réponse est obtenue de façon plus rapide par l'utilisateur d'un micro-ordinateur que s'il avait dû passer par un système de temps partagé où il n'est pas le seul utilisateur. Ne parlons pas des systèmes de traitement par lots qui donneraient des délais encore bien plus longs.

Le caractère « anglophone » du langage est secondaire dans la mesure où l'on peut, et c'est ce que nous ferons dans cet ouvrage, écrire en « BASIQUE » francophone moyennant des programmes de traduction appropriés très faciles à mettre en œuvre.

Ceci dit, BASIC reste un langage pratique, simple et disponible actuellement sur tous les micro-ordinateurs du marché. De ce fait, le nombre de programmes actuellement écrits dans ce langage augmente de façon appréciable de jour en jour !

CHAPITRE I

INTRODUCTION

Le langage BASIC est un langage de programmation évolué, encore appelé de haut niveau, car sa définition est indépendante de l'ordinateur ou de la machine sur lequel les programmes écrits dans ce langage seront exécutés. Avant d'étudier le langage proprement dit, il est donc nécessaire de présenter la structure de base des ordinateurs actuels et les concepts de base de la programmation.

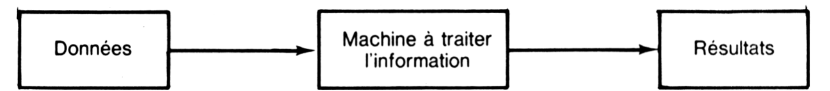
Dans ce chapitre nous présenterons donc successivement des généralités sur le traitement de l'information, sur la structure et le fonctionnement des ordinateurs, sur les algorithmes et les organigrammes. Ces notions ne sont pas propres au langage BASIC, mais constitue un préalable pour tout débutant ou futur utilisateur de ce langage. Le lecteur plus averti peut donc passer directement aux chapitres suivants qui présentent le langage proprement dit.

1. GENERALITES SUR LE TRAITEMENT AUTOMATIQUE DE L'INFORMATION

L'objet et l'intérêt de la programmation est de permettre de spécifier à une machine un certain travail à effectuer de façon automatique.

Pour cela il faut en général fournir à la machine les valeurs de certains paramètres que l'on appelle les *données*. Ensuite la machine effectuera un certain nombre d'opérations sur ces données, en suivant un certain schéma de fonctionnement qui lui aura été précisé, soit une fois pour toutes, soit à la demande (ce schéma de fonctionnement correspond à ce que l'on appelle le programme). Enfin, tout ceci ne présente d'intérêt que dans la mesure où la machine fournit un certain nombre de *résultats*.

Ainsi schématiquement on a :



Dans ce schéma l'homme n'intervient que pour alimenter la machine (données) et recueillir les résultats. (Il intervient également pour la conception du programme.)

1-1. Machines à programme figé

Il est important de souligner que le traitement automatique de l'information peut être effectué par des connexions mécaniques, électriques, optiques,... entre les différents éléments constituant la machine. On dit alors que le programme est figé (dans le cas d'une machine électronique on dira câblé). Des machines à programme figé sont forcément spécialisées pour un certain type de travail automatique (ainsi la machine à calculer de Pascal fut la première machine programmée mécaniquement pour effectuer des opérations arithmétiques). Parmi les machines à programme câblé, on peut citer tous les automates électroniques, depuis l'ascenseur ou le portillon automatique jusqu'aux différents appareils automatiques utilisés dans l'industrie (machines-outils, chaînes de fabrication, etc.).

La caractéristique principale des machines à programme figé est que le programme est codé d'une manière définitive et irréversible dans la structure même de la machine.

1-2. Machine à programme enregistré

Pour permettre plus de flexibilité, il faut avoir recours à des machines à *programme enregistré.* Le premier type de machine à programme enregistré fut décrit par Babbage (mathématicien anglais du XIXe siècle) à l'aide d'un programme extérieur à la machine qui est codé sur un support continu. Sur ce support sont donc enregistrées les différentes instructions constituant le programme.

L'avantage d'un tel type de machine est que l'on peut facilement changer le programme, et, par conséquent, la machine peut traiter différents types de problèmes : il suffit de réécrire le programme. On dit alors que la machine est universelle.

Ainsi paradoxalement, le concept de machine universelle n'est pas forcément synonyme de complexité : la machine de Türing,

(mathématicien anglais du XXe siècle), composée d'une tête de lecture et d'un ruban contenant le programme qui peut avancer ou reculer devant la tête de lecture, est capable d'exécuter toutes les opérations et tous les algorithmes exécutables sur les machines les plus modernes.

Le seul désavantage d'un tel type de machine est que la programmation est longue et fastidieuse et que, de plus, le temps d'exécution des programmes peut être très long pour des problèmes complexes.

Il a fallu le génie du mathématicien Von Neumann pour réaliser (1946), en quelque sorte, la synthèse entre les machines à programme figé interne et les machines à programme enregistré externe : l'idée fondamentale a été de considérer les programmes comme des données qui pouvaient être stockées à l'intérieur de la machine (dans une mémoire interne dans laquelle on peut enregistrer différentes informations qui sont soit des instructions de programmes, soit des données).

Cette idée et cette conception sont à la base des machines modernes ou ordinateurs. C'est seulement après la conception des ordinateurs que la programmation est devenue une discipline importante et que l'informatique est devenue une science et une technique distincte des mathématiques et de l'électronique.

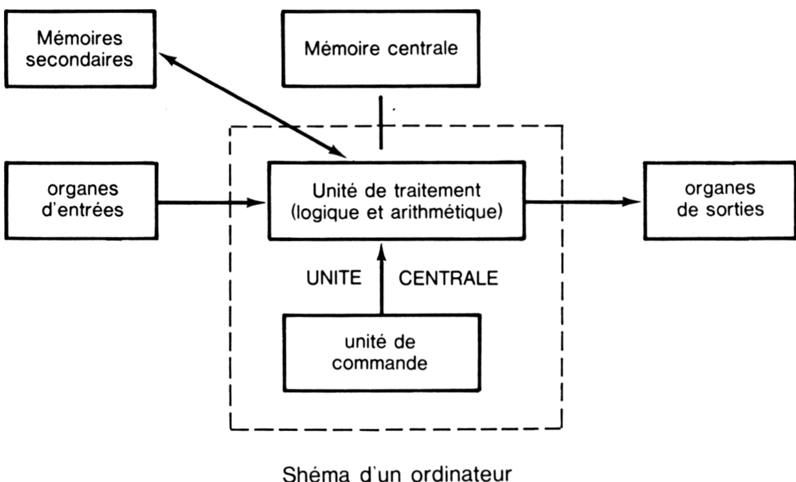
Remarques. – Dans les machines actuelles il existe une nouvelle notion de programme figé, en particulier lorsque l'on parle de « microprogrammation ». En fait, il s'agit de programmes qui ont été enregistrés dans des mémoires dites *mortes* que l'on ne peut modifier (ROM : Read only memory) ou que l'on ne peut modifier que très rarement. Ces techniques permettent de construire une machine relativement complexe à partir d'une « micromachine » ayant des instructions très élémentaires qui servent à l'écriture de microprogrammes qui réalisent des fonctions ou opérations plus élaborées. Nous ne nous étendrons pas ici sur ce type de programmation.

2. STRUCTURE ET FONCTIONNEMENT D'UN ORDINATEUR

Il n'est pas question ici de donner une structure détaillée d'un ordinateur, mais il est important de connaître l'outil que l'on va utiliser ; d'où la nécessité de présenter les différents organes constituant une machine à traiter l'information.

Les ordinateurs sont aussi appelés calculateurs digitaux, par opposition aux calculateurs analogiques. Un calculateur digital

travaille sur des données discrètes (non continues) qui sont codées par des nombres binaires (système de numération utilisant la base2).



Shéma d'un ordinateur

Dans ce schéma on distingue les organes suivants :

2-1. La mémoire centrale

C'est dans cette mémoire que sont contenus les programmes et les données. Du point de vue technologique, une mémoire principale est caractérisée par les éléments suivants :

- la *technologie* utilisée (autrefois tores magnétiques, maintenant semi-conducteurs : MOS (Metal Oxyde Semi-conductor), bipolaire, mémoire à bulles magnétiques, etc.) ;
- *le cycle mémoire de base,* qui indique le temps qu'il faut pour lire ou écrire un quantum d'information dans la mémoire. Dans les machines actuelles le cycle de base est de l'ordre de la microseconde, μs (les plus rapides sont de l'ordre de quelques centaines de nanosecondes).

Organisation de la mémoire

La plus petite information que l'on peut mémoriser est appelée le *bit* (contraction de *bi*nary digi*t*) et représente un chiffre binaire (0 ou 1). En français on peut utiliser le mot *eb* (élément binaire).

La capacité d'une mémoire pourrait être chiffrée en nombre total de bits, mais, pour faciliter le traitement et l'adressage des données contenues dans la mémoire, celle-ci a été divisée en « cellules » élémentaires de plusieurs bits que l'on appelle les *mots mémoires*, qui sont accessibles en bloc par une adresse qui représente la position de la cellule dans la mémoire. (Pour des raisons d'homogénéité toutes les cellules ont la même taille.) Ainsi une mémoire est elle aussi caractérisée par la taille de ses mots mémoires : 8 bits, 16 bits, 24 bits, 32 bits,... Bien que cette taille varie suivant les machines et les constructeurs, on trouve le plus souvent des tailles de mots qui sont des multiples de mots de 8 bits (appelés *octets*). Pour les petites machines (micro-ordinateur) le mot est le plus souvent de 8 bits ou de 16 bits.

Pour les miniordinateurs le mot de 16 bits est le plus souvent utilisé. Pour les machines plus puissantes les mots sont de 32 bits ou plus. Il faut cependant noter que les frontières entre micro, mini et gros ordinateurs tendent à se déplacer rapidement.

Taille d'une mémoire centrale

Enfin, étant donné une taille de mot, un autre paramètre caractérisant la mémoire centrale est le nombre total de mots contenus dans la mémoire. Cette taille totale est exprimée en multiple de K = 1 024 mots (1 024 = 2^{10}). Ainsi on parlera d'une mémoire de 4 K mots, de 16 K, 32 K,...). Ce paramètre est également utilisé pour indiquer la place occupée par un programme : ainsi si l'on a un programme occupant 20 K il ne sera pas possible de l'exécuter sur une machine ayant une mémoire centrale de 16 K.

2-2. L'unité de traitement

Elle se compose de deux blocs fonctionnels.

Actuellement les unités centrales peuvent être intégrées sur un seul circuit à haute intégration (LSI), appelé microprocesseur.

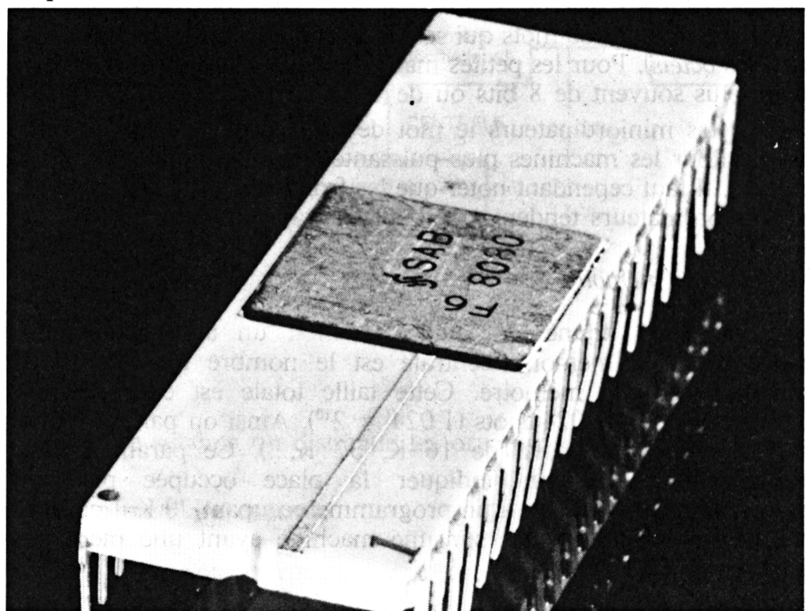
Elle constitue la partie active de la machine, celle qui traite les données pour fournir des résultats. Dans cette unité de traitement on trouvera toujours une unité de traitement arithmétique (addition, soustraction, multiplication, division,...) et logique (comparaisons et opérations logiques). Cette unité dispose également de mémoires spécialisées dans lesquelles sont effectués les traitements, et que l'on appelle les *registres* (accumulateur, registres d'index, pile, etc.).

L'unité de traitement (Central Processing Unit : CPU en anglais) est également caractérisée par un répertoire d'*instructions* qui est

utilisé pour indiquer le traitement à effectuer (on peut aussi les appeler des ordres, d'où le nom d'ordinateur).

Ces instructions sont élémentaires (ajouter deux mots, comparer deux mots, etc.) et sont les seules qui soient réellement exécutées par la machine.

Il est important de souligner que pour l'unité de traitement les instructions peuvent être considérées comme des données qui sont décodées et engendrent un certain nombre de commandes et d'opérations à l'intérieur de l'unité centrale.



un microprocesseur

L'unité de commande

Ce sont ces organes qui décodent les instructions et les transmettent à l'unité de traitement. Ces organes sont en relation avec la mémoire principale (pour lire ou écrire des informations dans la mémoire) et avec l'unité de traitement qui effectue les traitements proprement dits.

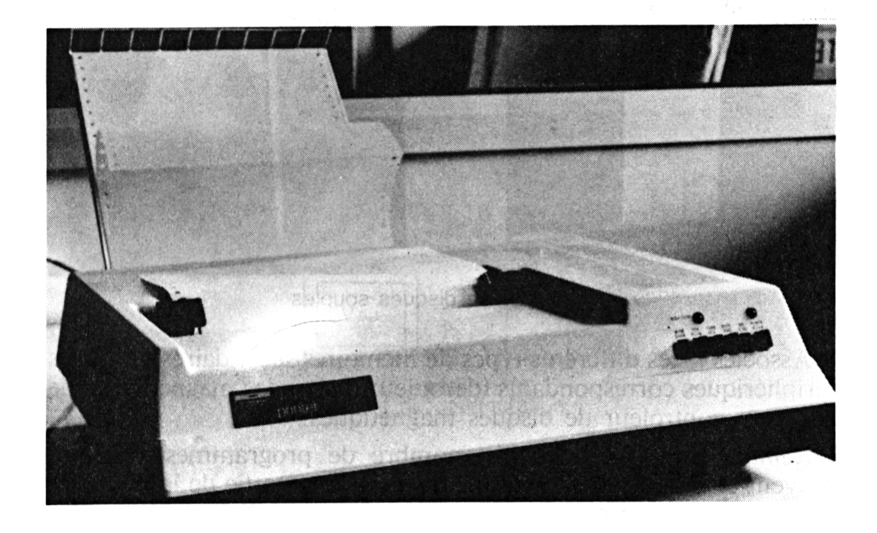
Ces organes de commandes servent également à décoder les ordres venant de l'extérieur ou allant vers l'extérieur (organes d'entrées-sorties).

L'unité de commande peut être réalisée à l'aide de logique électronique ou peut elle-même fonctionner à partir d'une mémoire morte contenant des micro-instructions. On dit alors que l'unité centrale est microprogrammée.

2-3. LES ORGANES D'ENTREES-SORTIES

Ces organes servent à faire communiquer la machine avec le monde extérieur et notamment à fournir à la machine les programmes et les données que l'on veut faire exécuter pour obtenir en retour des résultats. Ces organes sont ceux qui permettent à l'homme de commander et d'utiliser la machine.

Ces organes d'entrées-sorties sont aussi appelés organes *périphériques* de l'ordinateur. On peut citer par exemple : les lecteurs de cartes ou de rubans perforés, les perforateurs de cartes ou de ruban, les imprimantes, les téléscripteurs, les traceurs de courbe, les unités de visualisation, etc.

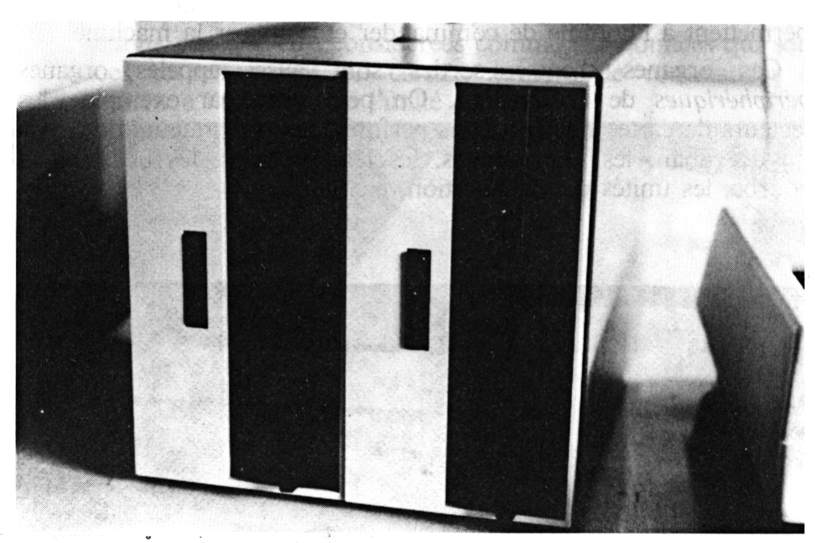


une imprimante

Dans cette catégorie il faut également ranger les unités de *mémoires secondaires* sur lesquelles on peut lire ou écrire des informations que l'on veut pouvoir conserver plus longtemps. En effet, du fait de son caractère universel et limité, il est exclu que la mémoire principale d'un ordinateur serve à stocker des informations (programmes ou données) plus longtemps que ne le nécessite l'exécution d'un programme. Il est donc possible d'utiliser des mémoires dites secondaires où l'on pourra stocker des informations de façon permanente ou quasi permanente.

Ces organes de mémoires secondaires sont de deux types :

- les mémoires à accès *séquentiel* (bandes magnétiques, cartes, rubans perforés, cassettes) ;
- les mémoires secondaires à accès *aléatoires* (disques magnétiques souples ou durs).



une unité de disques souples

Associés à ces différents types de mémoires secondaires, il y a les périphériques correspondants (dérouleur de bandes magnétiques, de cassettes, contrôleur de disques magnétiques).

Remarques. – Un certain nombre de programmes systèmes peuvent être stockés en permanence dans une partie de la mémoire centrale constituée par des mémoires mortes (ROM).

Dans certains cas, il est intéressant de considérer ces mémoires secondaires comme une extension de la mémoire principale. En utilisant des techniques plus sophistiquées (overlays, swapping, segmentation, pagination) on peut donner à l'utilisateur l'impression qu'il dispose d'une mémoire quasi illimitée. Nous ne détaillerons pas ici ces techniques dites de « mémoire virtuelle », mais il est important de savoir qu'elles existent et sont de plus en plus répandues.

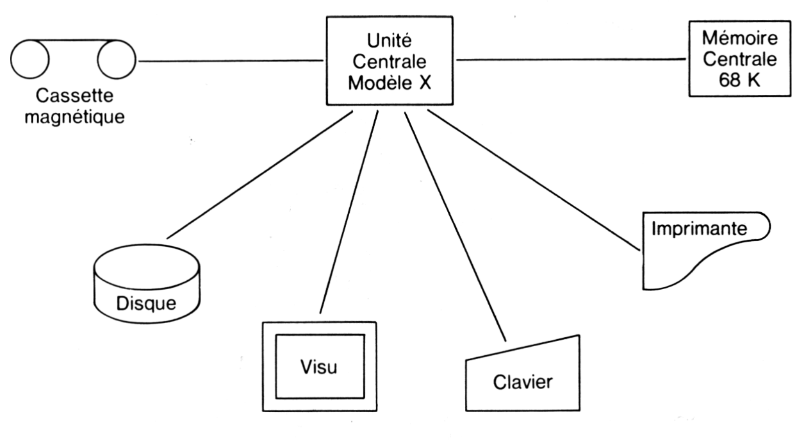
Il est à noter également que les organes d'entrées-sorties sont en général beaucoup plus lents que l'unité centrale (unité de traitement + organes de commandes), ce qui fait que des goulots d'étranglement se produisent fréquemment au niveau des organes entrées-sorties.

3. LES MICRO-ORDINATEURS

3-1. Notion de configuration

Lorsque l'on considère un ordinateur, il n'est pas toujours suffisant de connaître le type de machine utilisée, mais il est également très important de connaître la configuration disponible, c'est-à-dire l'ensemble taille mémoire, les périphériques et les extensions possibles.

Cette configuration peut être exprimée par un schéma du type :



Exemple de configuration

Ici le shéma correspond à une configuration type de micro-ordinateur.

Pour des ordinateurs plus puissants, cette configuration peut inclure beaucoup plus de périphériques en nombre et en variété.

3-2. Configuration des micro-ordinateurs

Un micro-ordinateur est un ordinateur dont l'unité centrale est un microprocesseur. Un microprocesseur est un circuit de haute intégration (LSI) qui contient toutes les fonctions d'une unité centrale.

Il existe différentes catégories de micro-ordinateurs :

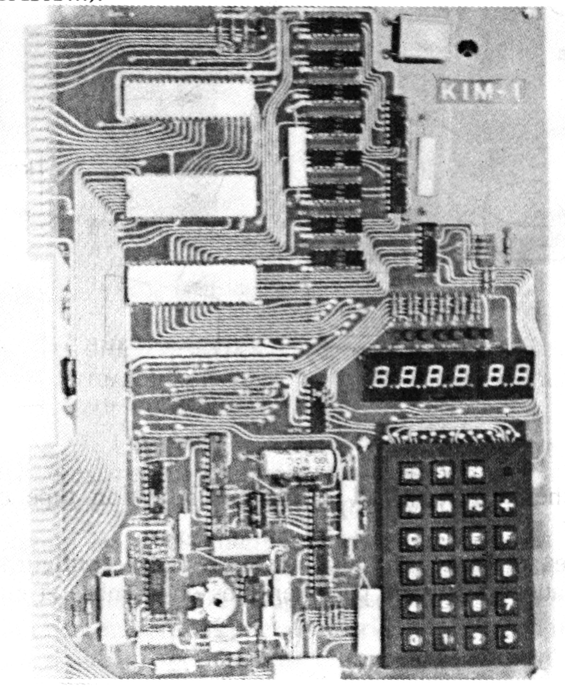
Les micro-ordinateurs en un seul circuit

Ce sont des microprocesseurs incorporant un programme figé en mémoire morte (ROM) et une petite mémoire vive (RAM). De tels circuits sont destinés à des applications très simples développées en grande quantité.

Ils ne sont pas encore programmables en langage évolué.

Les micro-ordinateurs sur une seule carte

Ce sont des cartes incorporant des circuits microprocesseurs, des circuits de mémoire morte (ROM), contenant des programmes, des circuits de mémoire vive (RAM) destinés à contenir des programmes et des données, des circuits d'interface d'entrées-sorties (PIO, WSART...).



une carte micro-ordinateur « KIM »

Jusqu'à récemment, ces cartes micro-ordinateurs n'étaient programmables qu'en langage machine, mais il existe actuellement des cartes micro-ordinateurs permettant de développer des programmes en BASIC (ainsi le micro-ordinateur SYM similaire à

celui présenté sur la figure ci-dessus permet, moyennant la connexion à un clavier approprié de programmer en langage BASIC. Il existe, d'autre part des cartes micro-ordinateurs permettant de programmer en langage évolué PASCAL (carte WD 9000).

Les micro-ordinateurs personnels

Ce sont des systèmes complets et compacts basés sur une carte micro-ordinateur, mais incorporant un clavier et une unité de visualisation avec possibilité de connexion à des périphériques standards telles que des unités de disques souples, des petites imprimantes, des cassettes magnétiques... Ce type de matériel peut être programmé en langage évolué et particulièrement en BASIC.



photo d'un système microordinateur « PET »

Les micro-ordinateurs professionnels

De tels sytèmes sont du point de vue interne basés sur des cartes de même type que les micro-ordinateurs personnels, mais sont plus complets et peuvent être connectés à plusieurs types de périphériques professionnels : unité de disques multiples, imprimantes rapides, bandes magnétiques, connexions à des réseaux, multiples terminaux...

En fait, cette catégorie de systèmes peut actuellement être confondue avec des systèmes de mini-ordinateurs. Ils disposent de logiciels sophistiqués : multiprogrammation, systèmes de gestion de fichiers, et peuvent être programmés en divers langages évolués : BASIC, FORTRAN, COBOL, PASCAL.

Ils peuvent être utilisés dans des applications commerciales ou industrielles en remplacement des systèmes mini-ordinateurs pour des coûts d'achat bien inférieurs.

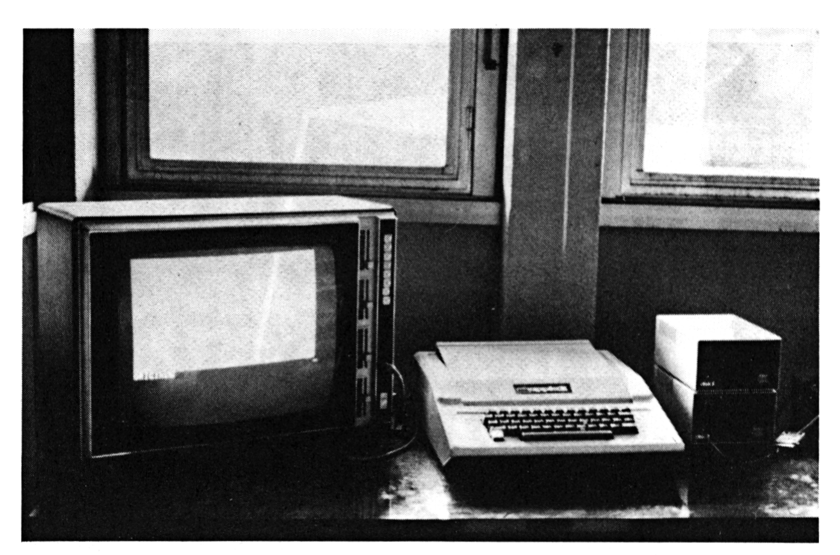


photo d'un système « APPLE »

Ce qui est commun à tous ces systèmes (à l'exclusion des micro-ordinateurs en un seul circuit) est qu'ils disposent tous actuellement du langage BASIC, ce qui montre donc l'importance de ce langage dans le domaine des micro-ordinateurs.

Remarque. – Sur le problème de savoir où s'arrête la classification de micro-ordinateurs nous avons vu que la dernière catégorie recouvre actuellement la catégorie des mini-ordinateurs. On peut donc dire que cette frontière n'a plus beaucoup de sens

dans la mesure où en 1979 on a vu apparaître des microprocesseurs 16 bits aussi puissants que des mini-ordinateurs de milieu de gamme. Ce qui est certain, c'est qu'actuellement, et de plus en plus, la programmation de ces micro-ordinateurs se fera à l'aide de langages évolués. Les logiciels sont de plus en plus sophistiqués, et, les utilisateurs étant de plus en plus nombreux, on peut prévoir que le développement de nouvelles applications se fera à l'aide de langages évolués. Cela aboutit à une disponibilité de produits logiciels de masse utilisables sur une grande variété de systèmes pour des coûts relativement faibles en raison du grand nombre d'exemplaires distribués.

C'est là un des caractères fondamentaux de l'arrivée des micro-ordinateurs sur le marché : le logiciel va pouvoir être développé sur une très grande échelle. Ceci n'exclut d'ailleurs pas la possibilité d'une nouvelle catégorie d'informaticiens « artisans » et fabricants de produits logiciels sur demande et diffusibles à des utilisateurs particuliers.

4. LA NOTION D'ALGORITHME

La tâche du programmeur est essentiellement d'exprimer les problèmes à résoudre sous forme d'algorithme et de les traduire sous forme de programmes dans un langage donné.

4-1. Historique et étymologie

Malgré les apparences (algorithme d'Euclide), le mot algorithme n'est pas un dérivé d'un mot grec ou latin, mais une contraction et une déformation du nom propre du mathématicien arabe Al Khwarismi, qui publia deux livres importants : l'un d'arithmétique, l'autre dont le titre *Kitab al-jabr wal muqabala* (Traduit par « action de faire passer et d'agencer les parties d'un tout ») est à l'origine du mot algèbre.

Lorsque le premier livre fut traduit en latin trois siècles plus tard, le titre donné au livre d'arithmétique fut *Algorismus*. En fait, à l'époque, cela désignait la méthode de numération de position utilisée maintenant en arithmétique, et qui avait été en fait découverte par les Hindous et transmise à l'Europe par les Arabes. Ainsi de par son origine le mot algorithme a quelque chose à voir avec la notion plus connue de procédé de calcul.

4-2. Essai de définition

La première définition du mot algorithme correspondant à son sens actuel est donnée par le mathématicien Markov :

« Tout ensemble de règles précises qui définit un procédé de calcul destiné à obtenir un résultat déterminé à partir de certaines données initiales. »

Les algorithmes obéissent aux trois propriétés suivantes qu'ils doivent nécessairement posséder :

a) ils sont constitués par un ensemble de règles précises et sont compréhensibles par tous ;
b) ils s'appliquent à des données qui peuvent varier dans une large mesure ;
c) ils tendent à la construction d'un résultat, que l'on obtient lorsque les données sont bien choisies.

Ainsi cette notion d'algorithme peut être assimilée à des notions plus communes de « méthode », « instructions », « technique », « processus », « recette », etc. Dans le cadre précis de l'informatique, cette définition est cependant légèrement insuffisante, car il faut y rajouter certains caractères opérationnels tels que :

- un algorithme doit être fini : c'est un ensemble fini d'instructions, chacune des instructions devant également se terminer dans un temps fini ;
- les processus intervenant dans un algorithme sont de nature discrète et non continue ;
- le fonctionnement de l'algorithme est déterministe : il doit toujours donner les mêmes résultats pour les mêmes données ;
- il existe un opérateur qui effectue ou exécute les instructions (cet opérateur peut être une personne, un appareil mécanique ou électronique, etc.) ;
- il existe un support de mémoire permettant de stocker non seulement les résultats intermédiaires et finaux, mais aussi les règles et les données.

La meilleure définition d'un algorithme valable actuellement dans le cadre de l'informatique a été donné par Knuth, dans son ouvrage sur l'*Art de la programmation :*

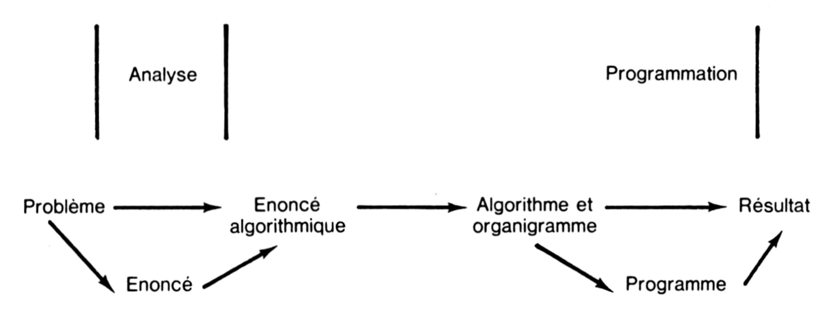
« C'est un ensemble de règles (ou instructions) ayant les cinq caractéristiques suivantes :

- il doit être fini et doit se terminer après un nombre fini d'opérations ;
- il doit être défini et précis : chaque instruction doit être définie sans ambiguïté ;
- s'il y a des données d'entrées, le champ d'application doit être précisé (ex. : nombres entiers, réels, négatifs,...) ;
- il doit posséder au moins un résultat (donnée de sortie) ;
- il doit être effectif : toutes les opérations doivent pouvoir être

effectuées exactement et dans un temps fini par un homme utilisant des moyens manuels.

4-3. Problèmes, algorithmes et programmation

Etant donné un problème concret, on commencera par définir un énoncé précis de ce problème (par ex. : calculer le plus commun diviseur de deux nombres), puis on passera à l'énoncé algorithmique qui indiquera les différentes étapes de l'algorithme sous forme d'organigramme ou de schéma de programme, qui sera ensuite programmé pour obtenir les résultats du problème posé.



Il faut remarquer qu'un problème donné ne conduit pas forcément à un algorithme unique. On dira qu'un problème est décidable s'il existe au moins un algorithme permettant de résoudre le problème. Il est important de savoir qu'il existe des problèmes qui ne sont pas décidables, c'est-à-dire pour lesquels il n'y a pas d'algorithme donnant une solution.

Il faut bien insister sur le fait que l'existence d'une solution pour un problème dans un cas particulier ou même sur plusieurs cas spécifiques n'implique pas que le problème soit décidable. Inversement, le fait de dire qu'un problème n'est pas décidable, n'implique pas qu'il n'ait pas de solutions, mais il implique qu'il n'y a pas d'algorithme qui permette de trouver ces solutions dans tous les cas où elles existent.

D'après ce que nous avons vu, un algorithme peut être considéré comme une suite d'actions à effectuer pour résoudre un problème. Ainsi un mode d'emploi peut être considéré comme la description de l'algorithme d'utilisation d'un appareil.

On peut d'ailleurs remarquer que l'apprentissage du calcul numérique ou même de l'analyse grammaticale et de l'orthographe à l'école primaire relève le plus souvent de méthodes algorithmiques. Ainsi, nous espérons avoir rassuré le lecteur que le concept d'algorithme effraie encore. Dans ce qui suit, un certain nombre

d'exemples simples vont être donnés : ils vont du plus simple au plus complexe, et l'on verra que ces différents types d'algorithmes se retrouvent en programmation.

Il faut cependant avertir le lecteur qu'il *n'y a pas de méthodes* pour découvrir un algorithme concernant un problème que l'on n'a pas encore résolu, soit personnellement, soit collectivement. La découverte d'un algorithme est donc un acte essentiellement *créatif* faisant appel non seulement à l'intelligence, mais également à l'intuition, et dans une certaine mesure à l'expérience.

D'autre part, un même problème peut être résolu avec plusieurs types d'algorithmes. Ainsi en est-il, par exemple, du problème de tri de nombres par ordre croissant ou décroissant. Il existe de nombreux algorithmes donnant une solution à ce problème.

Il faut donc distinguer trois types de problèmes :

- les problèmes que l'on sait résoudre pratiquement de façon manuelle ou intellectuelle dans tous les cas. Alors, l'analyse des méthodes utilisées pour résoudre ce problème doit permettre la formalisation d'un algorithme ;
- les problèmes pour lesquels on a déjà trouvé un algorithme. Il peut être alors intéressant d'en rechercher un autre plus simple ou aboutissant plus rapidement au résultat. Il s'agit d'un travail de recherche informatique ;
- les problèmes que l'on ne sait pas résoudre dans tous les cas ou pas du tout. Dans ce cas, l'analyse passe par une phase de recherche qui peut ne pas aboutir. Si le problème est indécidable, il n'y a pas d'algorithme. Si le problème est décidable, il faut d'abord le simplifier puis essayer de le généraliser. C'est alors un travail beaucoup plus difficile et dont le résultat est aléatoire.

4-4. Les différents niveaux de complexité des algorithmes

1° *La formule*

Ainsi, étant donné le problème : calculer le salaire brut d'un individu connaissant son salaire horaire et le nombre d'heures de travail.

On peut le résoudre par l'énoncé algorithmique suivant :

Soit H le salaire horaire.
Soit T le nombre d'heures.
Alors le salaire brut B = H × T.

L'algorithme correspondant est très simple, et peut se résumer de la façon suivante :

Données H, T.

B = H × T.
Résultat B.

Dans ce cas, une seule formule de calcul résume l'algorithme.

2° *La séquence de formules avec résultats intermédiaires :*

En compliquant légèrement le problème précédent on peut demander :

« Calculer le salaire net d'une personne en connaissant le salaire horaire, le nombre d'heures, et en incluant la retenue de la Sécurité sociale. »

Il faut alors rajouter dans l'énoncé algorithmique :

Soit P le pourcentage de retenue SS sur le salaire de base.

Alors la retenue est : R = B × P
et le salaire net : N = B − R.

Ou encore de façon plus condensée :

Données H, T, P.
B = H × T.
R = B × P.
N = B − R.
Résultat N.

Ici l'algorithme est une séquence de plusieurs formules avec des résultats intermédiaires tels que B et R.

On notera que l'ordre dans lequel se font les opérations correspondant aux formules n'est pas indifférent (ainsi B doit être calculé avant R et R avant N).

3° *Algorithme conditionnel*

En compliquant encore l'exemple précédent, on peut fixer un plafond pour la retenue de Sécurité sociale au-dessus duquel il n'y a plus de retenue.

Ainsi l'énoncé algorithmique doit être complété par :

Soit MAX.
B = H × T.
R = B × P.
Si = R > MAX alors R = MAX.
N = B − R.
Résultat N.

Ici on a introduit un test sur une condition consécutive à un calcul.

4° *Algorithme itératif*

Les algorithmes précédents ne fonctionnent que pour un seul jeu de données. Si l'on veut maintenant obtenir le calcul de la paie de 100 personnes en utilisant les algorithmes précédents, il faudra rajouter une itération sur l'ensemble des personnes. Ainsi l'algorithme deviendra itératif. En considérant I comme un numéro de séquence des individus on a :

Données communes P, MAX.
Pour I variant de 1 à 100.
Données H(I), P(I).
B(I) = H(I) × T(I).
R(I) = B(I) × P.
Si R(I) > MAX, alors R(I) = MAX.
N(I) = B(I) − R(I).
Résultat N(I).

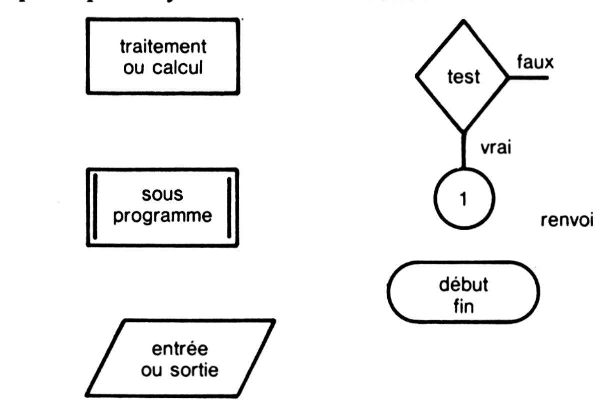
Ainsi sur ces exemples a-t-on vu l'ensemble des structures algorithmiques que l'on peut rencontrer dans tous les langages de programmation. Pour être complet, il faudrait y rajouter les algorithmes récursifs qui utilisent une définition faisant intervenir le résultat recherché dans les règles définies pour obtenir le résultat final.

Ainsi, par exemple, l'on sait que : $n! = (n-1)! \times n$, qui peut s'écrire FACT (N) = N × FACT (N − 1).

5. LES ORGANIGRAMMES

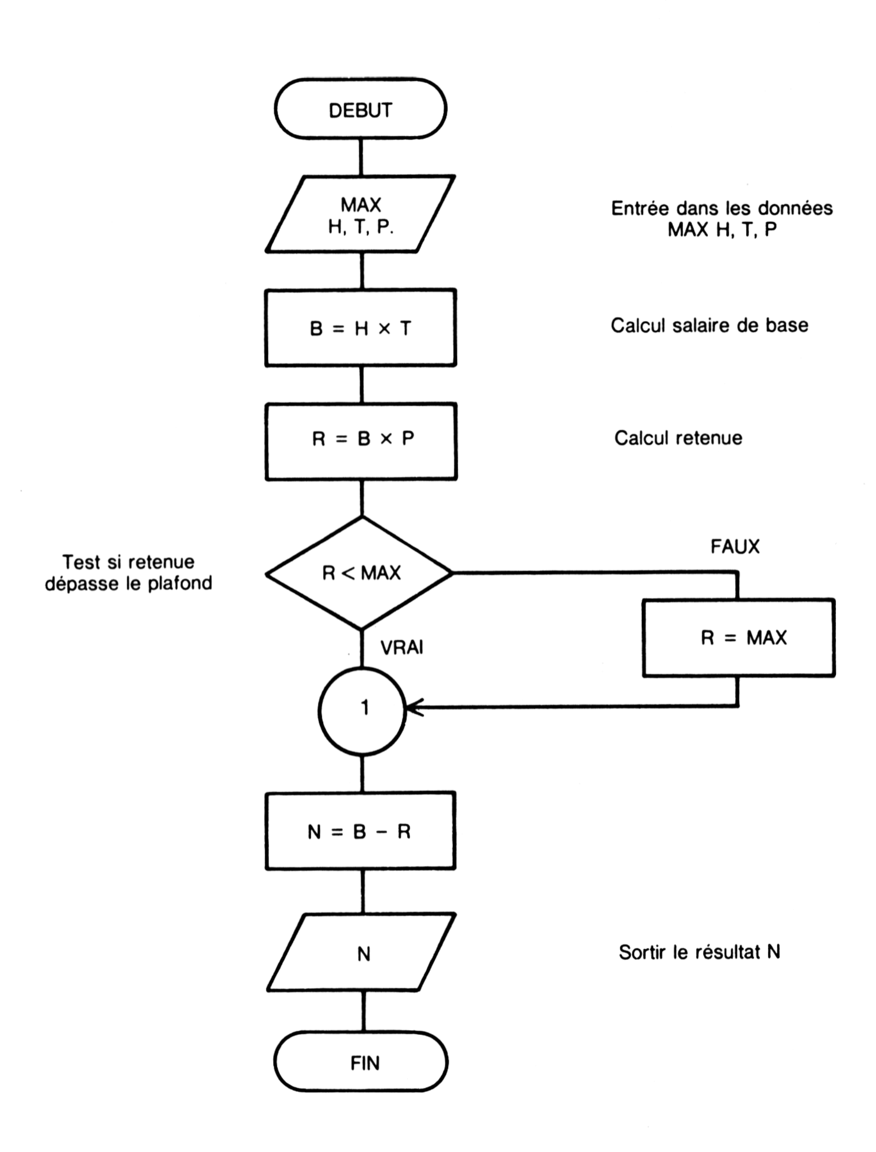
Il existe une méthode graphique de description des algorithmes qui consiste à décrire les différentes opérations sous forme d'un schéma indiquant les différents ordres et les différentes conditions qui sont traitées par l'algorithme.

Les principaux symboles utilisés sont :



Exemple :

Si l'on prend l'algorithme conditionnel précédent on aura par exemple :



Utilisation des organigrammes

En principe, dans un contexte industriel, la phase d'analyse d'un problème se termine par l'écriture d'un organigramme qui constitue en quelque sorte le « plan » de l'algorithme qui pourra être ensuite programmé. Ce schéma correspond en fait à la nécessité d'une certaine division du travail dans un contexte de production.

L'organigramme sert, dans ce cas, comme document de base à l'analyse organique servant de lien entre l'analyste et le programmeur. Il sert d'autre part à la documentation du travail de l'analyste et est théoriquement indépendant d'un langage de programmation donné. De ce fait, le même organigramme peut être programmé en plusieurs langages.

Cependant, depuis quelques années, les méthodes évoluent et l'organigramme n'est plus nécessairement le point de passage obligatoire d'un projet informatique. En effet, lorsque le problème est complexe, un organigramme ne tient plus sur une seule page, et il devient très difficile à suivre. D'autre part, il peut se révéler être faux à la phase de programmation, ce qui implique des mises à jour et des retours en arrière.

Pour toutes ces raisons associées à l'existence de techniques de programmation structurées, l'organigramme n'a plus la place qu'il avait il y a encore quelques années. Il faut noter cependant que, le langage BASIC n'étant pas très structuré, l'organigramme reste encore un outil qui peut aider à la programmation.

6. LE LOGICIEL ET LA PROGRAMMATION

Dans les paragraphes précédents, l'on a vu que les machines appelées ordinateurs ne pouvaient exécuter qu'un certain nombre limité d'actions élémentaires appelées instructions ou ordres.

La puissance des ordinateurs réside dans le fait qu'ils peuvent exécuter de longues séquences d'instructions élémentaires dans un temps très court (ainsi, si le cycle de base de l'unité centrale est de l'ordre de la microseconde, on pourra exécuter jusqu'à 1 million d'instructions par seconde). Il est bien évident que si l'on devait spécifier 1 million d'instructions différentes pour occuper l'ordinateur pendant une seconde, l'avantage retiré ne serait sans doute pas compensé par le temps passé pour effectuer ce travail fastidieux.

La programmation consiste précisément à trouver des méthodes itératives qui permettent de résoudre certains problèmes en n'ayant à spécifier qu'un nombre relativement limité d'instructions qui

devront être répétées plusieurs centaines, voire plusieurs milliers ou millions de fois par la machine.

6-1. Nécessité d'un codage et d'un langage symbolique

Le seul « langage » connu de la machine étant le langage binaire, il est indispensable de pouvoir spécifier les séquences d'instructions et les données sous une forme plus facilement manipulable par les hommes.

On aura ainsi recours à des notations symboliques qui permettront de décrire les actions à effectuer dans un langage symbolique.

Il existe plusieurs niveaux de symbolisme suivant que l'on utilise un langage spécifique ou non à la machine.

6-2. Notion d'instructions symboliques

Pour programmer le problème de calcul déjà vu en exemple, on pourrait imaginer un langage symbolique du type :

INSTRUCTION 1 : MULTIPLIER LA VALEUR DE LA MEMOIRE B AVEC CELLE DE LA MEMOIRE P ET RANGER LE RESULTAT DANS LA MEMOIRE R PUIS EXECUTER L'INSTRUCTION 2.

INSTRUCTION 2 : SI LE RESULTAT R EST INFERIEUR A MAX ALORS ALLER EXECUTER L'INSTRUCTION 4 SINON EXECUTER L'INSTRUCTION 3.

INSTRUCTION 3 : RANGER LA VALEUR MAX DANS R.

INSTRUCTION 4 : IMPRIMER LE MESSAGE : « RETENUE S.S. EST = » ; IMPRIMER LA VALEUR DE R.

Un tel langage présenterait l'avantage d'être compréhensible par tous, mais présente plusieurs inconvénients :

- pour chaque instruction élémentaire, il faut une phrase relativement longue ;
- il faut pouvoir décoder et traduire chacune des phrases de manière non ambiguë pour la machine (ainsi toute faute d'orthographe ou omission d'un paramètre peut rendre l'instruction incompréhensible pour la machine) ;
- pour un problème complexe, la suite d'instructions sera relativement longue, et il sera difficile de suivre l'enchaînement logique des opérations.

La première simplification possible est, d'une part, d'utiliser une organisation séquentielle des instructions : ainsi, dans l'exemple ci-dessus, on aurait pu éviter d'indiquer dans l'instruction 1 qu'il fallait ensuite exécuter l'instruction 2. Cette organisation séquentielle implicite est utilisée dans *tous* les langages de programmation : autrement dit, lorsque l'on lit ou écrit un programme de haut en bas les instructions sont exécutées les unes après les autres, sauf indication contraire (rupture de séquence).

Ainsi l'on voit que maintenant toute instruction symbolique est composée de trois types de paramètres : paramètre d'identification, paramètres indiquant la ou les opérations ou actions à effectuer, et paramètres opérandes.

Autrement dit, on pourrait programmer l'exemple précédent de la manière suivante :

1 MULTIPLIER B ET P ET RANGER LE RESULTAT DANS R
2 SI R< MAX ALLER A 4
3 SINON R = MAX
4 IMPRIMER « LA RETENUE SS EST = » ; IMPRIMER R

A partir de cet exemple très simple, on va voir que l'on peut aller dans deux directions pour définir un langage de programmation. La première direction consiste à vouloir rapprocher le langage symbolique des instructions réellement exécutées par la machine.

Langage assembleur : – Si la machine possède une instruction de multiplication appelée symboliquement MUL, et de soustraction appelée SUB, – des instructions d'appel et de rangement dans la mémoire (APM : appel mot mémoire, RTM : rangement mot mémoire) de branchement en cas de résultat négatif (BRN) et d'impression (IMP) ; on obtient le programme en langage machine :

étiquette	instruction	opérande	
I	APM	B	appeler le mot mémoire B
	MUL	P	multiplier par le mot mémoire P
	RTM	R	ranger dans le mot mémoire R.
	SUB	M	comparer à M
	BRN	N	se brancher à l'instruction N si M > R
	APM	M	
	RTM	R	faire R = M
N	IMP	MESSAGE	
	IMP	R	
MESSAGE	STOP	« RETENUE S.S. = »	
	CAR		

L'instruction CAR définit une suite de caractères.

Bien que ce ne soit pas le propos de cet ouvrage, cet exemple illustre ce que l'on appelle un langage symbolique d'assemblage (ou assembleur), dans lequel il faut préciser chaque opération élémentaire à effectuer. Cette solution présente l'inconvénient d'être particulière à la machine utilisée et d'être assez longue à programmer.

Langage évolué. – L'autre solution consiste à définir un langage indépendant de la machine sur laquelle on veut faire exécuter le programme. C'est vers cette solution que l'on s'oriente dans la suite du livre. Elle consiste à utiliser un langage dit universel ou évolué.

Ainsi, toujours dans le cas de l'exemple ci-dessus, on pourra décrire les opérations à effectuer à l'aide des instructions :

R : = B × P
SI R > MAX ALORS R : = MAX
IMPRIMER « RETENUE S.S. = » ; R

Ceci est très proche d'un langage universel tel que BASIC, ou PASCAL. Ainsi l'on voit les avantages d'un langage évolué ou universel par rapport à un langage d'assemblage : concision, possibilité d'utiliser des expressions proches de la formulation mathématique, compréhension quasi immédiate de ce que fait le programme.

Néanmoins, cela se traduit par la nécessité d'obéir à des règles de syntaxe précises qui permettent une traduction automatique du programme dans le langage de la machine. D'autre part, l'exécution du programme sera en général plus lente que si l'on avait utilisé un langage d'assemblage.

6-3. Notion de programme source et objet : compilation ou interprétation

L'utilisation d'un langage symbolique quelconque (assembleur ou évolué) nécessite que l'on traduise ce langage sous forme d'une séquence d'instructions machines.

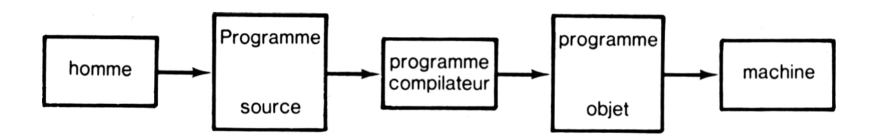
Il existe deux principes de traduction : la traduction globale, ou *compilation,* et la traduction instantanée au moment de l'exécution, ou *interprétation.*

a) *La compilation*

Le langage dans lequel est écrit le programme constitue le *programme source* et la forme sous laquelle il sera exécuté par la machine s'appelle le *programme objet.*

L'opération de traduction s'appelle la compilation et est opérée par un programme appelé le compilateur.

Ainsi, schématiquement :



pour le programme compilateur le programme source constitue les données et le programme objet les résultats. C'est le programme objet qui est exécuté par la machine.

Le programme compilateur est lui-même exécuté sur la machine et est en général écrit dans le langage d'assemblage de la machine, de manière à le rendre plus performant.

Lors de la compilation le programme compilateur pourra détecter et signaler un certain nombre d'erreurs de syntaxe ou de sémantique qui l'empêche de produire sans ambiguïté le programme objet. Dans ce cas, il faudra corriger ces erreurs, et un programme objet ne pourra être exécuté que lorsque toutes ces erreurs auront été éliminées.

Il est important de souligner cependant qu'un programme compilé sans erreur n'est pas forcément un programme exempt d'erreurs. En effet, le programme peut contenir des erreurs de logique qui ne peuvent être détectées lors de la compilation.

Pour terminer ce paragraphe, on signalera que pour un langage universel le programme source peut être traduit puis exécuté sur n'importe quelle machine disposant d'un compilateur correspondant au langage source (cela suppose une standardisation de ces langages). Par contre, l'exécution d'un programme objet ne sera possible que si les machines sont similaires ou compatibles et ont la même configuration.

Dans la pratique, cependant, il n'est pas rare que plusieurs compilateurs du même langage diffèrent légèrement d'une version à une autre, ce qui se traduit par une absence de transférabilité des programmes sources. Cela se résout soit en acceptant de se contraindre à utiliser les standards les moins souples, soit d'accepter d'avoir à modifier un certain nombre d'instructions lorsque l'on passe d'une machine à une autre.

b) *l'interprétation*

L'autre technique est de considérer que le programme source est traduit instruction par instruction au moment de son exécution.

Cette opération est effectuée par un interpréteur qui traduit chaque instruction en langage machine et l'exécute immédiatement.

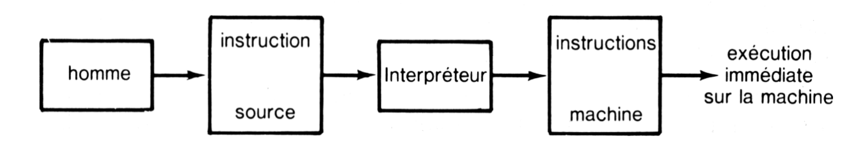


schéma d'un interpréteur

Dans ce cas, il n'y a pas de programme objet et les instructions sont traduites à chaque fois que le contrôle est donné à une instruction. La seule référence est alors le programme source.

Un langage interprété est en général *interactif*. C'est le cas du langage BASIC par exemple. On peut citer également le langage APL.

Ceci permet en particulier de développer et tester des programmes de façon très rapide sans avoir à passer par des phases de compilation, éditions de liens, chargement, exécution...

Avantages et inconvénients des langages compilés et interprétés

Les langages compilés présentent l'avantage d'être mieux structurés et, en général, plus lisibles. D'autre part, comme ils sont traduits en une seule fois, ils donnent un programme objet qui n'a plus besoin d'être traduit à chaque exécution. Le temps d'exécution global est donc beaucoup plus faible que celui d'un programme interprété.

Les langages interprétés permettent de développer et modifier un programme de manière interactive, ce qui facilite la mise au point des programmes. Ceci se paie au prix d'une exécution plus lente et, dans certains cas, de programmes mal structurés et difficiles à comprendre ou modifier.

Il faut cependant signaler qu'il existe des compilateurs BASIC, ce qui permet de pallier les inconvénients d'un langage interprété dans le cas de programmes de production... C'est le cas en particulier du C – BASIC utilisé dans les applications commerciales.

6-4. Les systèmes d'exploitation des ordinateurs

L'on a vu que l'utilisation de langages symboliques nécessitait l'existence d'au moins un programme de base : le compilateur ou

l'interpréteur. En fait, l'utilisation de langages évolués nécessite bien d'autres programmes dits utilitaires et de bibliothèques de sous-programmes. On trouve notamment tous les programmes réalisant les entrées-sorties, tous les programmes réalisant des fonctions mathématiques telles que racines carrées, logarithmes, exponentielles, etc.

Il est donc nécessaire d'avoir un ensemble de programmes constituant le système d'exploitation qui gère l'ensemble des périphériques et la séquence des opérations à effectuer depuis la lecture du programme source, la compilation ou l'interprétation, l'édition d'un listing, l'exécution du programme objet, jusqu'à l'édition des résultats. Toutes ces opérations sont réalisées sous le contrôle d'un programme que l'on appelle le *moniteur* ou encore le « système ». Dans certains cas, on parle également de programme superviseur, mais cette notion est en général réservée aux systèmes travaillant en temps réel (c'est-à-dire que chaque donnée venant de l'extérieur est traitée immédiatement ou dans un délai très court).

Il existe d'autre part des programmes utilitaires tels que les *éditeurs* permettant de rentrer un programme et/ou de le corriger ainsi que les programmes d'aide à la mise au point (« debugging »). Enfin, si l'on dispose de mémoires secondaires, des systèmes de gestion des fichiers sur disque (DOS) seront disponibles.

Monoprogrammation et multiprogrammation

Ces termes sont utilisés pour indiquer le type de système utilisé et non pas pour indiquer une méthode ou un type de programmation.

Ainsi, un système travaille en monoprogrammation si, à un instant donné, un seul programme est traité ou en cours de traitement par la machine (un seul programme est présent en mémoire centrale). Du fait de la grande différence de vitesse des périphériques et de l'unité centrale, il s'ensuit que, pendant l'exécution d'entrées-sorties, l'unité centrale est inactive, ce qui donne un rendement d'utilisation très faible lorsqu'il y a beaucoup d'entrées-sorties. Ceci est peu gênant dans le cas d'un système interactif ayant un temps de réponse raisonnable à l'échelle humaine.

Un système travaillant en multiprogrammation utilise le temps d'inactivité correspondant aux entrées-sorties d'un programme pour en exécuter un autre ou plusieurs autres. Ainsi augmente-t-on considérablement le rendement d'utilisation de l'unité centrale, mais en même temps on augmente la complexité du système. Les systèmes micro-ordinateurs de haut de gamme disposent de tels logiciels.

Dans la catégorie des systèmes de multiprogrammation, on a également les systèmes de temps partagé qui permettent un dialogue permanent avec la machine par l'intermédiaire de terminaux (machine à écrire ou écran de visualisation). Afin de permettre à chacun d'avancer dans ses travaux simultanément, chaque utilisateur est alternativement servi pendant une petite tranche de temps, d'où le nom de temps partagé. Ceci est utilisé sur des ordinateurs plus puissants.

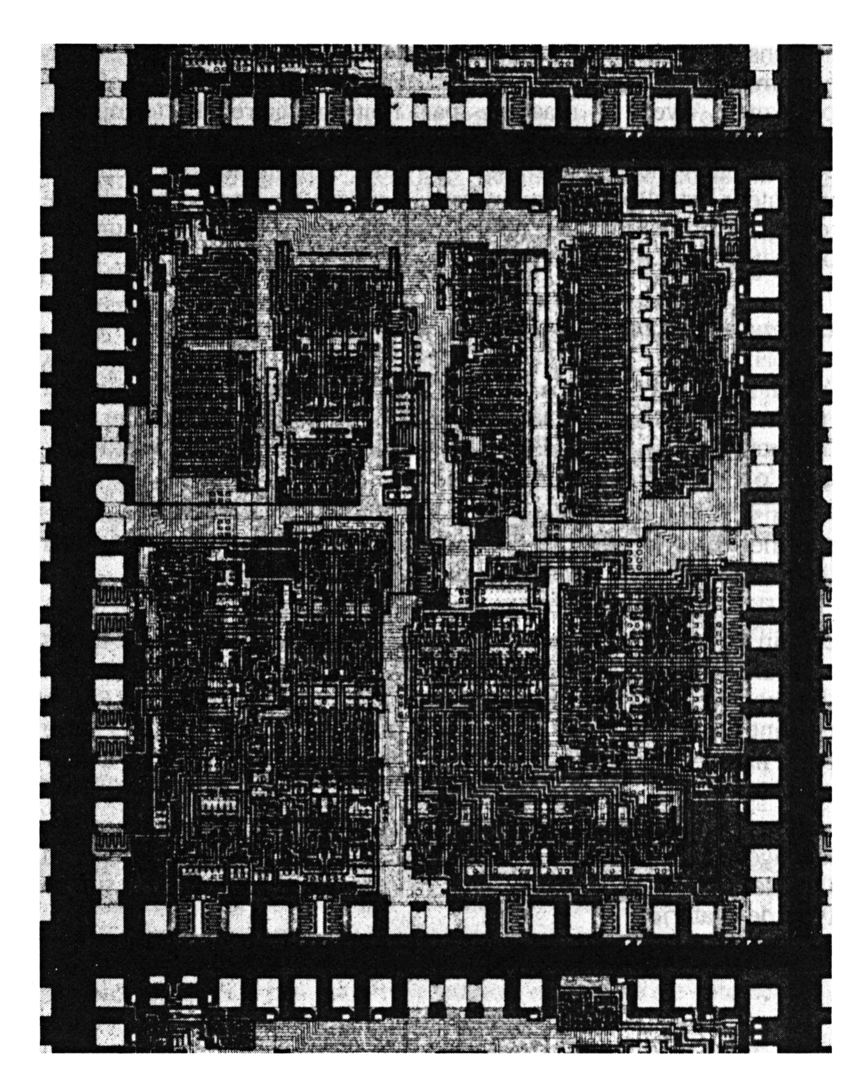
Langages de commande et de contrôle

Du fait de l'existence de ces systèmes d'exploitation plus ou moins complexes, il est également nécessaire de donner des instructions au système de manière à lui indiquer le type de travaux que l'on veut effectuer.

Ainsi a-t-on des langages de *commande* qui permettent de spécifier au système que l'on veut compiler un programme, puis d'obtenir un listing, et enfin exécuter un programme.

Sur les gros systèmes non interactifs, cela se traduit par un certain nombre de commandes exprimées sous forme de « cartes contrôles », que l'on doit placer avant et après le programme. Un des inconvénients majeurs de ces langages de commande est qu'ils ne sont pas standardisés et varient d'une machine à l'autre. D'autre part, ils sont souvent inutilement abscons et rigides quant à leur syntaxe.

Un des gros avantages des micro-ordinateurs est d'utiliser des langages de commande très simples et quasi transparents à l'utilisateur. Ceci facilite la mise en œuvre de programmes sur ce type de matériel.



une pastille (CHIP) de haute intégration

CHAPITRE II

GENERALITES SUR LE LANGAGE BASIC

Dans ce chapitre, nous présenterons de manière résumée les caractéristiques du langage. La version du langage à laquelle nous nous référons pour ce chapitre est le BASIC étendu, défini par le Dartmouth College (1960). Bien qu'il y ait des variations suivant les constructeurs, il s'agit pour l'instant du standard de base du langage.

Initialement, le langage a été développé comme un outil interactif pour l'initiation à la programmation. Des modifications y ont été apportées par rapport à la première définition (1960) pour assurer une meilleure transportabilité du langage sur des matériels divers et lui donner plus de souplesse. Dans ce chapitre nous donnerons les caractéristiques principales du langage.

Un programme est une suite de phrases écrites dans un langage défini par un alphabet et un ensemble de règles de formation des phrases dans ce langage. Les règles constituent la syntaxe ou la grammaire du langage.

Avant l'écriture d'un programme, il est nécessaire d'étudier le problème à résoudre et de trouver un *algorithme* permettant de définir de façon précise la résolution du programme. Un même problème peut être résolu par plusieurs algorithmes.

Nous avons vu qu'un algorithme est une suite d'instructions qui opère sur des données pour aboutir au bout d'un temps fini à un ou plusieurs résultats correspondant au problème à résoudre.

Ces suites d'instructions peuvent être simulées par l'esprit indépendamment de son écriture dans un langage de programmation ou de son exécution par une machine.

1. BASIC EST UN LANGAGE UNIVERSEL

Le langage BASIC dérive du langage de programmation FORTRAN, et dans cette mesure on peut le considérer comme un langage scientifique. Il faut noter cependant que BASIC dispose d'instructions de manipulation de caractères beaucoup plus pratiques que FORTRAN, ce qui le rend facile à utiliser pour des applications de gestion.

On peut donc dire que le champ d'application du langage est complet : problèmes scientifiques, problèmes de gestion, problèmes de traitement de textes, éducation..., etc.

D'autre part, contrairement à beaucoup de langages de programmation, la mise en œuvre de petits calculs est simple et directe : ainsi BASIC peut-il être utilisé pour effectuer des opérations de calculs élémentaires de façon aussi simple que sur une machine à calculer.

Enfin, les instructions d'entrée et de sortie de résultats sont également très simplifiées et plus souples que dans beaucoup d'autres langages de programmation. Un programme est composé d'une suite d'instructions que l'on peut appeler des « phrases » du langage.

Nous allons étudier les règles de formation de ces instructions. Des exemples seront donnés pour chaque type d'instructions.

2. BASIC EST UN LANGAGE INTERACTIF ET CONVERSATIONNEL

Le langage BASIC est le premier langage de programmation interactif développé il y a bientôt vingt ans, à une époque où les ordinateurs n'étaient accessibles qu'en traitement par lots (« batch processing »).

En ce sens, il a constitué un progrès par rapport aux langages compilés, car il a permis le développement et l'utilisation de programmes en mode conversationnel à partir d'un terminal avec clavier et imprimante. Par la suite BASIC est devenu le langage des petits systèmes conversationnels sur miniordinateurs ou sur les plus gros systèmes de temps partagé (« time sharing »).

De ce fait, BASIC est également le premier langage utilisant un interpréteur. Il existe beaucoup de dérivés de BASIC, notamment le FOCAL ou le langage MUMPS, proposés par le constructeur DIGITAL. En France, il faut également signaler le langage d'enseignement de la programmation dans le secondaire LSE, qui est assez proche du BASIC.

Le langage est basé sur une conception d'instructions correspondant à une ligne frappée sur un clavier de machine à écrire. La fin d'une ligne est représentée par le caractère Retour à la ligne. Il se dégage donc de la conception de langages basés sur la carte perforée tels que FORTRAN ou COBOL avec des zones bien délimitées et rigides.

Dans la suite du livre nous noterons le caractère Retour à la ligne par le symbole ®.

Lorsque la machine répond, nous utiliserons le caractère □ pour indiquer que le texte qui suit est envoyé par la machine.

BASIC est également un langage permettant d'utiliser un ordinateur comme machine à calculer.

3. BASIC LANGAGE DIRECT DE CALCUL ARITHMETIQUE

En supposant que l'on soit sous le contrôle d'un interpréteur BASIQUE (BASIC en français) et que l'on tape au clavier la séquence :

IMPRIMER 7 + 5 ® ou PRINT 7 + 5 ®

la machine répondra sur un écran de visualisation ou sur une machine à écrire : le résultat :

□ 12.

De même si l'on frappe :

IMPRIMER 7 − 5 ® ou PRINT 7 − 5 ®

le résultat imprimé ou visualisé sera :

□ 2.

Les autres opérations possibles sont bien sûr la multiplication et la division :

```
IMPRIMER 3 * 6 ®            PRINT 3 * 6 ®
    □ 18
IMPRIMER 10/3 ®             PRINT 10/3 ®
    □ 3.333333
```

On voit dans ce dernier exemple que le BASIC, dans sa version étendue, permet de faire des calculs immédiats sur des chiffres décimaux ou fractionnaires. La virgule est remplacée par le point pour indiquer la séparation entre la partie entière et décimale.

Il est également possible avec le BASIC étendu de demander l'impression de plusieurs opérations :

Exemple :

```
IMPRIMER     2 + 7,     7/2,     5 * 3 ®
    □          9         3.5       15
```

Dans ce cas, les résultats sont espacés par des blancs et sont imprimés dans l'ordre où les opérations ont été spécifiées.

Dans les exemples ci-dessus, le langage a été utilisé pour effectuer des calculs arithmétiques simples, mais il est également possible d'obtenir des résultats de fonctions mathématiques usuelles.

Ainsi, si l'on frappe :

IMPRIMER ABS (– 12) ®　　　　PRINT ABS (– 12) ®

on obtient :

□ 12, c'est-à-dire la valeur absolue de – 12.

De même

IMPRIMER RAC (2) ®　　　　PRINT SQR (2) ®

donnera la racine carrée du nombre entre parenthèses

□ 1.414...

ou encore

IMPRIMER LOG (2) ®　　　　PRINT LOG (2) ®

□ 0.30103...

Les autres fonctions mathématiques usuelles sont également disponibles (voir ci-dessous).

Jusqu'à présent, le langage n'offre pas d'avantages par rapport à une calculatrice de poche. On va maintenant voir qu'il est possible de faire imprimer du texte, ce qui n'est pas possible avec une calculatrice habituelle. En effet, lorsque l'on sort un résultat affiché ou imprimé sans autre indication, on ne sait plus à quoi correspond ce résultat.

Tout d'abord on peut faire imprimer des chaînes de caractères.

Une chaîne de caractères est une suite de caractères frappés sur clavier et qui est délimitée par des caractères spéciaux indiquant le début et la fin de la chaîne. Les caractères utilisés sont les doubles apostrophes ou guillemets : "

Ainsi " BONJOUR " est une chaîne de caractères.

On peut demander à la machine de l'imprimer en frappant :

IMPRIMER " BONJOUR " ®　　　　PRINT " BONJOUR " ®

□ BONJOUR

Ce qui est plus intéressant, c'est de pouvoir combiner l'impression d'un texte avec un calcul.

Ainsi peut-on écrire l'instruction :

IMPRIMER " LA TVA EST EGALE A ", 40 * 0.176 ®

□ LA TVA EST EGALE A 7.04

Nous nous arrêterons pour l'instant à ces exemples d'utilisation de BASIC comme langage de calcul direct. En effet, la puissance du langage vient de la possibilité d'écrire des suites d'instructions constituant un programme.

3-1. La notion d'instruction en BASIC

On a vu dans les paragraphes précédents des exemples d'instruction directe constituant une phrase du langage : ce texte comprend des mots spécifiques du langage (mots clés ou mots réservés), des signes de ponctuation, des noms spécifiés par le programmeur, des chiffres, des opérateurs, des chaînes de caractères.

Cet ensemble obéit à un certain nombre de règles de composition et de *syntaxe.*

Apprendre un langage de programmation, c'est donc avant tout apprendre sa syntaxe, mais ce n'est bien sûr pas suffisant pour savoir résoudre un problème !

En BASIC, une instruction comprend un numéro d'identification suivi du texte de l'instruction. Dans un même programme, deux instructions ne peuvent avoir le même numéro. Ces numéros d'identification s'appellent aussi des étiquettes.

Exemple :

```
10      IMPRIMER " LA SOMME DE 4 + 5 EST ", 4 + 5.
(10     PRINT " LA SOMME DE 4 + 5 EST ", 4 + 5).
```

10 est l'étiquette.

- IMPRIMER (PRINT) est un mot clé du langage.
- " LA SOMME DE 4 + 5 EST " est une suite de caractères à imprimer.
- 4 + 5 est une suite de chiffres séparés par un opérateur + indiquant qu'il faut faire une addition.

Dans ce cas, il s'agit d'une instruction très simple destinée à imprimer un message associé à un résultat de calcul arithmétique (ici l'addition de 4 + 5).

Le résultat de cette instruction se traduira par une impression et un calcul aboutissant à :

□ LA SOMME DE 4 + 5 EST 9

L'intérêt d'une telle instruction est bien sûr très limité, dans la mesure où il faut la réécrire à chaque fois que l'on veut faire une simple addition !

3-2. La notion de variable

De la même façon que l'on apprend à compter sur des chiffres et que l'on passe ensuite à des symboles pour représenter des variables susceptibles de prendre plusieurs valeurs, en programmation on utilise également des variables pour représenter des valeurs (ou données) qui changeront à chaque exécution du programme.

Ainsi peut-on considérer deux variables X et Y pour représenter deux nombres entiers ou réels et définir la somme par une expression :

$$S = X + Y$$

En BASIQUE, on peut ainsi écrire :

```
10      SOIT S = X + Y
20      IMPRIMER " LA SOMME DE X + Y EST ", S
```

Soit, en BASIC :

```
10      LET S = X + Y
20      PRINT " LA SOMME DE X + Y EST ", S
```

Ces deux instructions ne sont pas suffisantes pour obtenir un résultat, car X et Y ont des valeurs inconnues.

3-3. La notion de donnée

Si l'on veut effectuer comme précédemment une opération sur des nombres, il faut donner à X et Y des valeurs.

On peut bien sûr rajouter les instructions :

```
 1      SOIT X = 4
 2      SOIT Y = 5
10      SOIT S = X + Y
20      IMPRIMER " LA SOMME DE X + Y EST ", S
30      FIN
```

```
 1      LET X = 4
 2      LET Y = 5
10      LET S = X + Y
20      PRINT " LA SOMME DE X + Y EST ", S
30      END
```

Ceci constitue une suite d'instructions tout à fait légale qui donnera un résultat correct.

Cependant, si maintenant l'on désire effectuer l'opération 124 + 385, il faut réécrire les instructions 1 et 2. L'avantage d'avoir introduit des variables est perdu par le fait que l'on assigne à ces variables des valeurs constantes dans le programme.

Pour avoir plus de souplesse, il faut donc considérer X et Y

comme des variables qui contiendront des valeurs que l'on donnera à la machine au moment de l'exécution du programme. C'est ce que l'on appelle des données.

Une *donnée* est donc une valeur que l'utilisateur (et non pas le programmeur) donnera à la machine au moment de *l'exécution* du programme.

Ainsi, en BASIQUE on peut demander l'entrée des données destinées à être manipulées par le programme.

Les instructions 1 et 2 sont alors remplacées par une instruction ENTRER (INPUT).

```
1     ENTRER X, Y
10    SOIT S = X + Y
20    IMPRIMER " LA SOMME DE ", X, " + " , Y, " EST ", S
30    FIN
1     INPUT X, Y
10    LET S = X + Y
20    PRINT " LA SOMME DE ", X, " + ", Y, " EST ", S
30    END
```

Au moment de l'exécution du programme la machine demandera l'entrée des données X et Y sous forme d'un point d'interrogation : ?

Si l'on répond 4 suivi d'un retour chariot (touche RETURN), cela correspondra à la valeur de X, la machine renverra alors un deuxième point d'interrogation et si l'on répond 5 suivi d'un retour chariot, alors l'exécution du programme continuera et donnera le résultat escompté :

LA SOMME DE 4 + 5 EST 9

L'avantage est que cette fois si l'on réexécute le programme en rentrant les données 124 et 385 on obtiendra :

LA SOMME DE 124 + 385 EST 509

sans avoir à modifier le programme.

Cette dernière version constitue donc véritablement un programme général qui permet de faire l'addition de deux nombres quelconques.

3-4. La documentation des programmes.

Un programme très simple comme celui présenté ci-dessus ne nécessite pas d'explication supplémentaire. Par contre, lorsque ces programmes comprennent plusieurs dizaines voire centaines ou milliers d'instructions, il devient difficile pour un utilisateur (y compris pour un programmeur) de s'y retrouver et de comprendre

le programme. Dans ce cas on peut rajouter des instructions qui ne sont pas exécutées par la machine, mais qui servent à documenter et / ou expliquer ce que fait chaque partie du programme.

Cette instruction est précédée par un mot clé REM (Remarque) suivi de texte libre expliquant ce que l'on veut préciser.

Exemple

```
10    REM ENTREE DES DONNEES
20    ENTRER X, Y
30    REM CALCUL DE LA SOMME
40    SOIT S = X + Y
50    REM IMPRESSION DU RESULTAT
60    IMPRIMER "LA SOMME DE", X, " + ", Y, "EST", S
70    FIN
```

3-5. Utilisation d'un système interpréteur BASIC

Nous présentons ici succinctement les principales commandes qui seront détaillées à la fin du chapitre.

Le BASIC est un langage interprété. Bien qu'il existe dans certains cas des compilateurs BASIC, nous supposerons qu'il s'agit d'un système interpréteur. Sur la plupart des micro-ordinateurs l'interpréteur est stocké dans une mémoire morte (ROM), et dès la mise sous tension le contrôle est donné à l'*éditeur*. Le système répond alors par PRET (READY) ou un message similaire.

L'éditeur est un programme qui permet de rentrer des lignes de texte et en particulier des lignes d'instructions BASIC correspondant à un programme.

Les éditeurs disponibles sont plus ou moins sophistiqués. En général, ils permettent de rentrer des lignes de texte qui sont validées une fois que la touche retour à la ligne ® (RETURN) a été enfoncée. Dans certains cas l'éditeur numérote les lignes, mais le plus souvent les numéros doivent être introduits par l'utilisateur ce qui donne plus de souplesse.

a) *Cas d'un nouveau programme*

Si l'on désire rentrer un nouveau programme on frappera alors une commande spéciale NOUVEAU (NEW) ® juste après le message PRET. Si l'on frappe directement une instruction au clavier, elle sera prise en compte dès que la touche RETOUR aura été frappée. Ainsi, si la machine vient d'être mise en route, il n'est pas nécessaire de frapper de commande.

b) *Cas d'un programme déjà en mémoire*

Dans ce cas chaque nouvelle instruction tapée au clavier sera insérée dans le programme existant. En particulier, si l'instruction est étiquetée avec un numéro déjà existant, elle remplacera l'instruction ancienne correspondant à ce numéro.

c) *Liste d'un programme*

Un programme peut être créé et modifié en plusieurs étapes. Si l'on désire obtenir la dernière version, il suffit d'utiliser une commande appelée LIST, qui permet d'obtenir la liste des instructions classées dans l'ordre croissant des numéros d'étiquettes.

Ceci permet de vérifier que le programme est complet et correct avant de le lancer.

d) *Lancement d'un programme*

Une fois que le programme est jugé correct, il peut être exécuté en utilisant une commande qui demande l'exécution et l'interprétation du programme. Cette commande s'appelle en général MARCHE (RUN), et l'interpréteur commence alors à interpréter les instructions et à les exécuter. C'est seulement à ce moment que l'utilisateur sait si son programme est correct du point de vue syntaxique et du point de vue logique.

Exemple :

Si l'on veut effectuer la séquence d'opération

$$15 + 7 \text{ et } 15 - 7$$

on écrira :

```
10      IMPRIMER 15 + 7
20      IMPRIMER 15 - 7
```

suivi de MARCHE (RUN)

Il y aura alors interprétation et exécution de l'instruction 10, puis de l'instruction 20, ce qui donnera les résultats :

```
□ 22
   8
```

suivi de PRET (READY), ou d'un message équivalent.

e) *Suppression et insertion d'instructions*

Pour *supprimer* une instruction, il suffit de frapper le numéro de l'instruction suivi d'un RETOUR.

Pour *insérer* une instruction, il suffit de définir une instruction avec un numéro tel qu'il s'insère entre deux instructions déjà définies.

Ceci explique qu'il est prudent de numéroter les instructions d'un programme de 10 en 10, de manière à pouvoir insérer ultérieurement des instructions.

4. BASIC EST UN LANGAGE ALGORITHMIQUE

Les langages permettant de transcrire des algorithmes exécutables sur une machine sont dits algorithmiques. BASIC est donc un langage algorithmique.

Exemple :

La notion d'algorithme n'est pas nouvelle, et l'on connaît le fameux algorithme d'Euclide pour calculer le PGCD (plus grand commun diviseur de deux nombres). Ainsi, peut-on l'exprimer par les instructions I_1 à I_5 :

I_1. Soient deux nombres entiers a et b $(a > b)$.
I_2. Diviser a par b et soit r le reste $(0 \leqslant r < b)$
I_3 : Si r est nul alors l'algorithme est terminé et b est le PGCD.
I_4 : Sinon remplacer a par b et b par r
I_5 : Continuer en I_2.

On peut déjà remarquer sur cet algorithme que les instructions I_1 à I_5 ne sont pas du même type. I_1 est une hypothèse ou affirmation. I_2 est une instruction de calcul arithmétique. I_3 est une instruction conditionnelle (si... alors... sinon...). I_4 est une instruction de manipulation de variables. I_5 est un ordre inconditionnel.

Ces différents types d'instructions se retrouvent en effet dans tous les langages de programmation algorithmiques.

4-1. Déclarations et instructions exécutables

Un algorithme est composé de deux types d'instructions :

- des instructions de description des *données* encore appelées *déclarations* ou (définitions) ;
- des instructions de description des *actions* à effectuer qui sont appelées *instructions* exécutables.

BASIC est un langage qui possède très peu d'instructions de *déclarations,* car la plupart des déclarations sont implicites. Dans l'exemple précédent, on peut remarquer que a et b, considérées comme données en I_1, deviennent des variables en I_4.

Ainsi, dans un programme, les données sont représentées par *les valeurs de variables*. Ces variables sont manipulées dans les instructions exécutables.

4-2. Identificateurs et mots clés

Dans un langage de programmation, ces variables sont représentées par des *identificateurs :* c'est le nom par lequel on désigne cette variable.

Les règles de formation de ces identificateurs sont très simples, ce qui laisse une grande liberté de choix. En BASIC, ces noms sont constitués d'un caractère alphabétique, suivi ou non de chiffres.

Par contre, dans tout langage de programmation, existent un certain nombre de *mots clés* ou mots réservés, qui ont un sens bien défini et ne peuvent être utilisés que dans certaines structures syntaxiques du langage. La liste exhaustive des mots clés BASIC est donnée plus loin.

4-3 Les déclarations en BASIC

En BASIC, il n'y a pas d'instructions de déclaration explicites donnant le type d'une donnée ou variable.

Le type d'une donnée est l'ensemble des valeurs que peut prendre une variable. On a vu qu'en BASIC il existe deux types élémentaires de données et de variables : les données numériques et les données de type *chaînes de caractères.*

Exemple

Si une variable prend des valeurs numériques, son type sera défini comme entier ou réel suivant les valeurs numériques traitées.

En BASIC, les données sont soit de type *scalaire,* c'est-à-dire représentées par un ensemble de valeurs élémentaires, soit de type *structuré,* permettant de définir des ensembles de valeurs arrangées sous forme de liste ou de tableau. Il faut noter cependant que les tableaux sont limités à deux dimensions.

a) *Déclarations de type scalaire*

Il existe deux types de données : les *nombres* et les *chaînes de caractères.*

Ces deux types de données n'ont pas à être déclarées explicitement, mais c'est l'identificateur de la variable qui détermine le type de celle-ci. Ainsi, si l'identificateur est suivi du caractère $ il sera considéré comme chaîne de caractères. De même, s'il est suivi du caractère % la variable sera considérée comme représentant un nombre entier.

b) *Déclarations de type structuré*

Il existe deux types de structures de données : les tableaux de valeurs numériques et les tableaux de chaînes de caractères.

Dans une structure de tableau, tous les composants sont de même type. Un élément du tableau est sélectionné par un indice qui doit être un entier. Cet indice permet d'accéder à n'importe quel élément du tableau dans le même temps. Dans le cas de déclaration de type structuré, il faudra donner les dimensions du tableau.

4-4. Les instructions exécutables

a) *Instructions de calcul*

Comme on l'a vu dans l'exemple d'algorithme donné ci-dessus, les instructions exécutables sont de deux types : les instructions de calcul et les instructions de transfert ou de contrôle du programme.

Pour un langage de programmation évolué, il faut y rajouter des instructions d'entrée/sortie. D'autre part, dans le langage BASIC, comme dans tous les langages de programmation actuels, les instructions de calcul sont des instructions *d'affectation.*

Une instruction d'affectation est une instruction comprenant deux membres : un membre gauche et un membre droit, séparés par un opérateur d'affectation (=). Le membre droit est celui qui précise le calcul à effectuer, tandis que le membre gauche indique la variable utilisée pour mémoriser le résultat des calculs précisés dans le membre gauche.

Exemple :

F = A * B + 3

Ici, le calcul effectué est $a \times b + 3$ et le résultat est affecté (mémorisé) dans la variable F.

Nous reviendrons plus en détail sur ces instructions et sur les règles de syntaxe associées dans les chapitres suivants.

Programmation d'un algorithme de calcul simple

Nous reprendrons les exemples d'algorithmes donnés dans le premier chapitre.

Les données H et T représentent le nombre d'heures et le tarif horaire : ce sont des données numériques. On peut donc les considérer comme des variables numériques appelées H et T. Le résultat est également numérique et est appelé B.

10	ENTRER H, T	10	INPUT H, T
20	SOIT B = H * T	20	LET B = H * T
30	IMPRIMER B	30	PRINT B
40	FIN	40	END

Si l'on doit calculer la retenue de Sécurité sociale et le salaire net, on rajoutera simplement une donnée P et deux instructions de calcul permettant de calculer la retenue et le salaire net.

Ici, cependant, la donnée P n'est pas modifiée. On peut donc la considérer comme une constante.

```
10      SOIT P = 0.04          pourcentage de 4 %
20      ENTRER H, T
30      SOIT B = H * T
40      SOIT R = B * P
50      SOIT N = B - R
60      IMPRIMER B, R, N
70      FIN
```

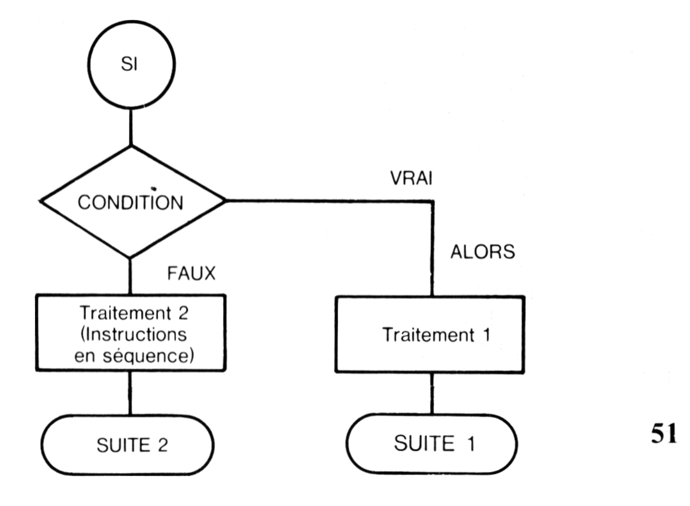
En BASIC, on aurait :

```
10      LET P = 0.04
20      INPUT H, T
30      LET B = H * T
40      LET R = B * P
50      LET N = B - R
60      PRINT B, R, N
70      END
```

b) *Les instructions de transferts*

Les instructions de *transfert ou de contrôle* sont des instructions qui permettent, soit d'effectuer des *tests* de certaines conditions renvoyant vers des traitements différents du programme, soit de donner le contrôle à une instruction particulière du programme.

Dans cette catégorie on retrouve les *instructions de test* ou *conditionnelles* qui ont une structure très commune dans tous les langages de programmation : le SI... ALORS... (IF... THEN...) représentée par l'organigramme suivant :



L'autre instruction de transfert est celle qui provoque un branchement inconditionnel à une instruction du programme : c'est l'instruction ALLER A (GO TO).

Exemple de programmation d'un algorithme conditionnel

Nous reprendrons l'exemple n° 3 du chapitre Ier.

La donnée supplémentaire est la valeur du plafond, que l'on appellera M.

Il faut donc tester si la retenue est supérieure à ce plafond, ce qui se fait par une instruction de test SI... ALORS.

On obtient alors le programme :

```
10      SOIT P = 0.04
20      SOIT M = 200
30      ENTRER H, T
40      SOIT B = H * T
50      SOIT R = B * P
60      SI R> M ALORS SOIT R = M
70      SOIT N = B - R
80      IMPRIMER B, R, N
90      FIN
```

Soit en BASIC :

```
10      LET P = 0.04
20      LET M = 200
30      INPUT H, T
40      LET B = H * T
50      LET R = B * P
60      IF R> M THEN LET R = M
70      LET N = B - R
80      PRINT B, R, N
90      END
```

On aurait pu également programmer cet algorithme en utilisant l'instruction ALLER A

```
 10     SOIT P = 0.04
 20     SOIT M = 200
 30     ENTRER H, T
 40     SOIT B = B * P
 60     SI R> M ALORS ALLER A 100
 70     SOIT N =  B - R
 80     IMPRIMER B, R, N
 90     STOP
100     SOIT R = M
110     ALLER A 70
120     FIN
```

Cette programmation est également correcte, mais plus lourde. Elle a été utilisée ici pour montrer l'utilisation du branchement inconditionnel, qui est nécessaire dans certains cas.

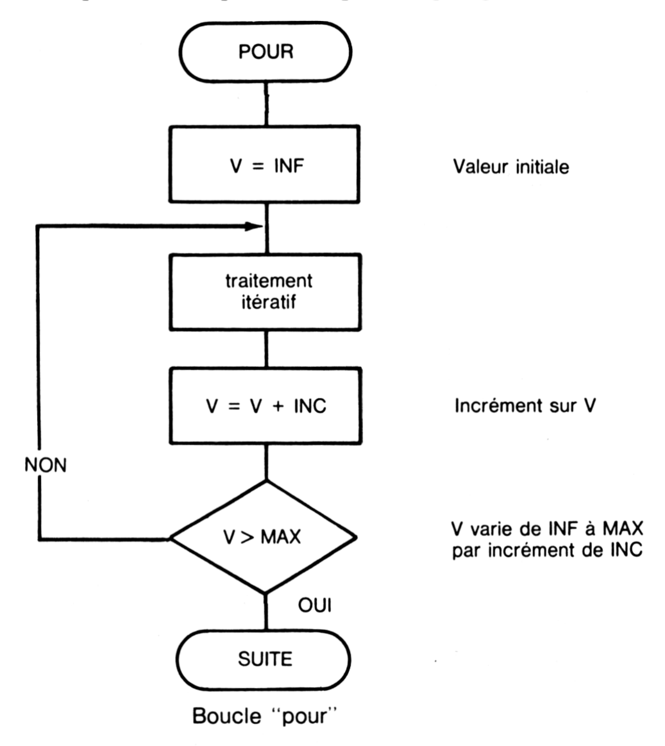
On voit également ici l'instruction STOP, qui permet d'arrêter l'exécution du programme de manière à ne pas continuer sur l'instruction 100 après avoir imprimé les résultats recherchés.

c) Les structures d'algorithmes itératifs ou répétitifs

Cette dernière structure permet de faire un certain nombre d'itérations en utilisant une variable de contrôle dont on fixe les bornes de variations. Il s'agit de la boucle de programme très courante dans tous les langages évolués (boucle POUR, "DO", "FOR")...

Exemple : POUR V = INF A MAX FAIRE...

Cette structure peut être représentée par l'organigramme.



Boucle "pour"

En BASIC, la boucle POUR peut également s'effectuer en utilisant des incréments négatifs. Dans ce cas, la variable de contrôle décroît d'une valeur maximale à une valeur minimale.

Exemple de programmation d'algorithme répétitif

Dans ce cas, les données et variables sont les mêmes, mais l'on demande à la machine d'effectuer les mêmes calculs sur 10 individus.

On obtient alors le programme :

```
10      SOIT P = 0.04
20      SOIT M = 200
30      POUR I = 1 A 10
40      ENTRER H, T
50      SOIT B = H * T
60      SOIT R = B * P
70      SI R> M ALORS SOIT R = M
80      SOIT N = B - R
90      IMPRIMER B, R, N
1000    SUIVANT I
110     FIN
```

On voit ici un exemple d'instruction POUR qui permet de faire exécuter dix fois les instructions 40 à 90.

L'instruction « SUIVANT I » indique que l'on passe à l'itération suivante, c'est-à-dire que la valeur de I est incrémentée de 1 à chaque itération et ceci jusqu'à ce que I atteigne la valeur 10.

En BASIC on aurait :

```
10      LET P = 0.04
20      LET P = 200
30      FOR I = 1 TO 10
40      INPUT H, T
50      LET B = H * T
60      LET R = B * P
70      IF R> M THEN LET R = M
80      LET N = B - R
90      PRINT B, R, N
100     NEXT I
110     END
```

Remarques. – On a vu dans ce paragraphe les caractéristiques essentielles du langage, mais bien sûr cela ne représente qu'un aperçu des possibilités du langage. Il existe en effet d'autres instructions, en particulier la possibilité de définir des sous-programmes, de traiter des chaînes de caractères, des tableaux et dans certains cas des possibilités de traitement de matrices. Dans les chapitres suivants, nous reviendrons donc plus en détail sur ces intructions, mais auparavant il est utile de préciser quelques commandes nécessaires à l'utilisation d'un système BASIC.

5. LES LANGAGES DE COMMANDE SUPPORTANT UN SYSTEME BASIC

Nous avons vu qu'il existe des systèmes d'exploitation pour gérer les différentes ressources d'un ordinateur. C'est ce que l'on appelle le moniteur ou superviseur.

Lorsqu'on utilise un ordinateur disposant d'un interpréteur BASIC, il est nécessaire de pouvoir dialoguer avec ce système d'exploitation à l'aide de commandes connues du moniteur. Les commandes ne font pas partie du langage lui-même et varient d'un système à l'autre. Dans cette mesure, les commandes que nous allons présenter sont les commandes minimum nécessaires pour pouvoir développer et exécuter des programmes.

Ces commandes correspondent d'une part à des mots ou phrases à base de mots clés et sont prises en compte lorsque l'on frappe un retour à la ligne ou retour de chariot (touche "RETURN" noté ® dans ce livre).

Il existe d'autre part des commandes d'édition qui permettent de faire des modifications du texte déjà rentré. Ces commandes d'édition correspondent à des caractères spéciaux.

5-1. Les commandes au système

Ces commandes permettent de définir différents modes de fonctionnement du moniteur.

En général, à la mise sous tension du système ou après avoir donné un certain nombre de codes d'identification sur un système de temps partagé, le moniteur indiquera qu'il est prêt à recevoir des commandes en indiquant : PRET (READY) ou tout simplement un caractère spécial tel que], > ou tout autre caractère suivant les sytèmes (c'est ce que l'on appelle un « prompt » dans le jargon informatique).

A partir de ce moment l'utilisateur a le choix entre la frappe d'instructions BASIC ou de commande au moniteur. Les unes étant différenciées des autres par le fait que les instructions BASIC commencent toujours par un numéro d'instruction (sauf pour les instructions directes telles que PRINT, qui peuvent dans une certaine mesure être considérées comme des commandes au moniteur).

a) L'initialisation d'un nouveau programme

Si l'on vient de commencer une session au terminal, ou de mettre la machine sous tension, la mémoire disponible pour l'utilisateur est

libre et on peut d'emblée frapper des instructions. Si, au contraire, l'on a déjà travaillé et que l'on désire écrire un autre programme, on frappera une commande indiquant que l'on commence un nouveau programme.

C'est la commande :

NOUVEAU (NEW) ®

Cette commande efface la mémoire du contenu de tout programme antérieur.

b) Le listage d'un programme ou d'instructions.

Lorsque l'on veut visualiser les instructions du programme actuellement en mémoire il suffit de spécifier une commande de listage :

LIST ®

Ceci liste l'ensemble du programme.

Si l'on désire lister une seule instruction on utilisera la commande :

LIST *n* ®

n étant un numéro de ligne.

Ainsi, LIST 140 ® listera l'instruction 140.

Dans le cas où l'on désire lister un ensemble de lignes consécutives on utilisera la commande :

LIST *n*1 – *n*2 ®

*n*1 étant la première instruction à lister et *n*2 la dernière.

Dans certains cas LIST – *n*1 ® permet de lister jusqu'à la ligne *n*1 ou bien LIST *n*1 – ® permet de lister à partir de la ligne *n*1 jusqu'à la fin du programme.

Exemple :

LIST 100 – 300 ® permettra de lister toutes les instructions entre 100 et 300.

Remarque. – Dans certains cas, le séparateur utilisé est une virgule ou un autre caractère.

c) *Effacement d'instructions*

Cette opération se fait différemment suivant les systèmes. Dans le cas où existe une commande il s'agit d'une commande d'effacement :

EFFACER *n* ®
(DELETE *n* ou DEL *n*)

Sur certains systèmes, il suffit de spécifier un numéro de ligne suivi de ®, ce qui a le même effet.

Exemple : 10 ® efface la ligne 10.

Si l'on dispose d'une commande d'effacement on aura aussi la possibilité d'effacer plusieurs lignes consécutives :

DEL *n*1, *n*2 ®

efface toutes les instructions de *n*1 à *n*2.

d) *Exécution d'un programme*

Un programme qui se trouve déjà en mémoire peut être lancé en frappant la commande MARCHE :

RUN ®

Dans le cas où l'on veut faire démarrer le programme à une instruction précise, on spécifiera :

RUN *n*1 ®

ce qui lancera le programme à partir de l'instruction *n*1.

Exemple :

RUN 100 ®

lance le programme à partir de l'instruction 100.

e) *Sauvegarde d'un programme sur mémoire secondaire*

Cette opération suppose que le système dispose d'une mémoire externe de type cassette magnétique ou disque magnétique (souple ou dur).

Dans ce cas, on utilise la commande SAUVER (SAVE), suivi d'un nom de programme ou nom de fichier.

Ainsi, par exemple :

SAVE PROG 1 ®

permet de mémoriser le programme actuellement en mémoire et de lui donner le nom PROG 1 sur le support de mémoire secondaire utilisé.

Cette commande peut varier suivant les systèmes utilisés, mais le mot clé SAVE est le plus souvent utilisé.

f) *Chargement d'un programme à partir d'une mémoire secondaire.*

C'est l'opération inverse de la précédente. Elle permet de lire un programme se trouvant dans un fichier sur mémoire secondaire et de le charger en mémoire. C'est la commande CHARGER (LOAD) qui permet de le faire.

Exemple :

LOAD PROG 2 ®

permet de charger en mémoire centrale le programme appelé PROG 2.

Remarques. – Si l'on dispose de plusieurs unités de mémoire secondaire, il faudra également préciser le numéro de l'unité sur laquelle se trouve le programme à charger.

Pour la syntaxe exacte de ces commandes, il faut bien sûr se référer aux manuels d'utilisation de ces machines. Dans le cas de systèmes de temps partagé, il faudra également préciser d'autres paramètres tels que la version du programme.

g) *La réinitialisation des variables d'un programme*

Lorsqu'on exécute un programme plusieurs fois, les contenus de certaines variables sont conservés d'une exécution à l'autre. Si l'on veut réinitialiser les valeurs de toutes les variables d'un programme, la commande qui permet de le faire est la commande REMISE A ZERO

- CLEAR OU CLR ®

Cette commande permet donc de réexécuter un programme dans les mêmes conditions initiales.

h) *La continuation d'un programme*

La dernière commande importante d'un système BASIC est celle qui permet de continuer l'exécution d'un programme après un arrêt défini dans le programme (c'est une instruction STOP).

Dans ce cas on utilisera une commande CONT ®.

Cette commande aura donc pour effet de continuer le programme au point où il en était lors de la rencontre de la dernière instruction STOP.

i) *L'arrêt d'un programme en cours*

Ceci se fait soit par une touche spéciale appelée STOP ou RESET, soit à l'aide d'une combinaison d'un caractère de contrôle (CONTROL) noté *c* et associé en général à la lettre C (CONTROL C noté C*c*).

5-2. Les commandes d'édition

Lorsque l'on veut modifier les instructions d'un programme, il est nécessaire de pouvoir disposer de commandes permettant de faire de l'édition des instructions déjà rentrées.

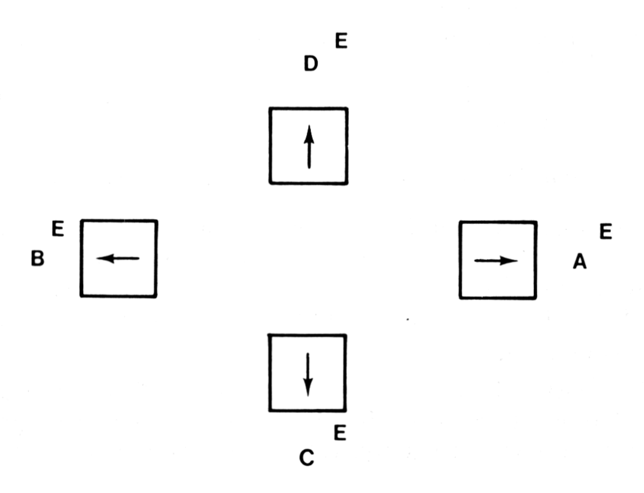
Or, on a déjà vu que l'insertion d'une ligne consiste simplement à donner un nouveau numéro compris entre les numéros des lignes où doit se faire l'insertion : ceci explique qu'il est prudent dans un premier temps de numéroter les instructions de 10 en 10, de manière à pouvoir facilement insérer de nouvelles instructions.

L'effacement d'une ligne a déjà été vu. Par contre, ce qui est très utile, c'est la possibilité de modifier une instruction sans avoir à la refrapper complètement. Pour cela il existe deux possibilités : soit l'on utilise les commandes d'un éditeur qui n'est pas spécifique au système BASIC (ceci est le cas sur les systèmes de temps partagé qui proposent plusieurs types de langages) soit l'on fait appel à l'éditeur du BASIC.

Dans la plupart des systèmes micro-ordinateurs, les fonctions d'édition sont disponibles directement au clavier grâce à des possibilités de mouvement d'un curseur sur un écran de visualisation.

a) *Mouvement du curseur*

Les mouvements du curseur sont possibles grâce à des touches spécialisées permettant de le faire se déplacer en haut, en bas, à gauche, à droite. Ceci est en général représenté par des touches représentant des flèches associées à ces mouvements :



Lorsque ces touches ne sont pas disponibles, il est possible d'obtenir le même effet en utilisant une touche de contrôle appelée en général ESC (ESCAPE), associée à une Lettre. On la notera E. Le code ASCII en est 27.

Le plus souvent, on a les combinaisons suivantes :

- A^E permet de faire le mouvement du curseur à droite ;
- B^E permet de faire le mouvement du curseur à gauche ;
- C^E dirige le curseur vers le bas ;
- D^E dirige le curseur vers le haut.

b) *L'insertion et l'effacement de caractères*

Ces fonctions ne sont pas toujours disponibles directement sur le clavier. Si elles existent, elles sont libellées en général INST (INSERER) et DEL (EFFACER).

Le plus souvent, il s'agit de commandes à l'éditeur, de touches fonctions spéciales ou associées à des combinaisons de touches de contrôle et d'un caractère.

c) *Autres commandes disponibles sur certains systèmes*

Certaines commandes concernent l'effacement d'écran et le retour au début de l'écran. Si ces commandes sont disponibles sur le clavier, elles s'appellent alors ERASE (effacer l'écran) ou CLR. Le retour du curseur s'appelle en général "HOME".

Dans certains systèmes (système "APPLE" notamment), on trouve les fonctions suivantes associées à la touche ESC (notée E). Cette touche doit être combinée à un autre caractère :

- E^E efface une ligne depuis la position du curseur jusqu'à la fin de la ligne ;
- F^E efface l'écran depuis la position du curseur jusqu'au bas de l'écran ;
- C^E efface tout l'écran et effectue le retour du curseur en haut de l'écran.

d) *Les commandes associées à la touche de CONTROLE*

Ces commandes sont plus ou moins standardisées, et on les retrouve sur plusieurs systèmes. Elles associent la touche CONTROL associée à une autre lettre sur des claviers ASCII standard (standard Americain utilisé pour le codage des caractères).

On a notamment :

- CONTROL B noté B^C suivi d'un retour ® qui transfère le contrôle au moniteur BASIC et détruit tout le programme existant (c'est en fait équivalent à la commande NEW) ;

- CONTROL G noté G^C permet d'émettre un "BIP" sur un haut-parleur incorporé ou sur une sonnette ;
- CONTROL H noté H^C permet d'effectuer un retour en arrière du curseur en effaçant les caractères rencontrés ;
- CONTROL V noté V^C permet d'effectuer une avance du curseur vers la droite en recopiant les caractères rencontrés ;
- CONTROL J : J^C permet de sortir une ligne ;
- CONTROL X : X^C permet d'effacer la ligne sur laquelle se trouve le curseur.

CONCLUSION

Dans ce chapitre, nous n'avons fait qu'aborder les caractéristiques générales du langage et avons présenté principalement les commandes directes du langage ainsi que les commandes moniteurs susceptibles d'être disponibles sur des systèmes BASIC. Il faut insister sur le fait que, les commandes ne faisant pas partie du langage proprement dit, il n'y a pas de standards stricts, mais les fonctions présentées existent en général toujours sous une forme similaire. Nous conseillons donc au débutant d'essayer de retrouver sur le système utilisé ces différentes fonctions et de les tester avant de passer aux chapitres suivants, qui concernent le langage proprement dit. En effet, ces fonctions sont essentielles pour gagner du temps au moment de l'écriture de programmes ou de leur modification.

CHAPITRE III

LES ELEMENTS DE BASE DU LANGAGE

INTRODUCTION

Un langage, et en particulier un langage de programmation, est défini à l'aide d'un *alphabet.*

A partir de cet alphabet on peut constituer des mots en utilisant des règles de construction que l'on appelle des règles *lexicales*. Les mots d'un langage sont reconnus par le fait qu'ils obéissent à ces règles et sont compris entre des séparateurs : le caractère blanc noté ƀ ou □ ou △ est en général utilisé. Un certain nombre de caractères peuvent avoir une signification précise dans ce langage : on les appelle alors *opérateurs.*

De même, un certain nombre de mots ont une signification précise dans ce langage : ces sont des *mots clés.*

L'assemblage des mots, de séparateurs et de mots clés constituent des *phrases* du langage. On les appelle aussi des *instructions* dans le cas des langages de programmation.

Il est évident que toutes les phrases que l'on peut constituer ne sont pas correctes. Leur construction obéit à certaines règles que l'on appelle les règles de *syntaxe* qui constituent la *grammaire* du langage.

L'étude d'un langage de programmation passe donc par la définition arbitraire d'un alphabet et de règles lexicales. En ce qui concerne les règles syntaxiques, elles ne sont pas totalement arbitraires, car elles doivent permettre de définir des algorithmes.

Cependant, ces règles syntaxiques doivent être choisies pour éliminer toute ambiguïté dans la construction de phrases correctes du langage.

Un langage de programmation doit donc être non ambigu, et, en pratique, cela se traduit par le choix d'un certain type de grammaire que l'on appelle grammaire *non contextuelle.*

De telles grammaires sont, d'autre part, caractérisées par certaines propriétés qui permettent de reconnaître facilement une phrase correcte d'une phrase incorrecte. Ceci facilite l'écriture des programmes traducteurs (compilateur ou interpréteur). Ainsi les instructions peuvent être traduites sans référence au contexte et sans ambiguïté.

Ceci se traduit par des contraintes au niveau de la programmation : ainsi, l'oubli d'un seul caractère ou d'un seul mot rendra une instruction incorrecte.

1. L'ALPHABET DU LANGAGE;

L'alphabet du langage comprend tous les caractères utilisables dans les phrases de ce langage. C'est aussi ce que l'on appelle les éléments terminaux du langage.

Il faut remarquer qu'un langage de programmation informatique utilise un jeu de caractères beaucoup plus étendu que l'alphabet usuel.

Dans le langage BASIC, l'alphabet comprend :

alpha-numérique
- les lettres majuscules : de A à Z (26 caractères) ;
- le caractère blanc : b̸ (ou △ ou □) (1 caractère) ;
- les chiffres décimaux : de 0 à 9 (10 caractères) ;
- les symboles spéciaux qui comprennent les opérateurs, les signes de ponctuation et les séparateurs.
  - \+ – / *↑ opérateurs arithmétiques
  - \> < = opérateurs relationnels
  - % ( ) $ " séparateurs
  - . , : signes de ponctuation.

Soit un alphabet de 53 caractères.

Le caractère Retour à la ligne ® ne fait pas partie de l'alphabet proprement dit, mais indique la fin d'une ligne de programme.

Remarque : Dans les nouveaux BASIC, développés sur micro-ordinateur, un certain nombre de caractères spéciaux sont utilisés pour définir des opérations graphiques sur écran cathodique ou téléviseur. Nous ne donnons pas ces caractères ici, car ils sont spécifiques à chaque constructeur. Il faut noter cependant que ce jeu de caractères supplémentaires permet de développer des programmes de visualisation graphiques ou semi-graphiques qui étendent les possibilités du langage.

2. LES REGLES DE FORMATION DES MOTS DU LANGAGE

Comme on l'a déjà vu, les mots du langage sont de trois types :

- les *identificateurs* choisis par le programmeur ;
- les *constantes* définies par le programmeur ;
- les *mots clés* ou mots réservés au langage.

2-1. Les identificateurs de variables numériques

Les règles de formation des identificateurs sont très simples en BASIC standard : ce sont des mots de deux caractères dont le premier est une lettre et le second un chiffre qui est facultatif.

Il s'agit là d'une limitation particulière aux « anciens » BASIC, qui donne donc une possibilité de 286 noms de variables différentes dans un même programme.

Les identificateurs possibles sont donc :

A, A_0, A_1 . . . . A_9
B, B_0, B_1 . . . . B_9
•
•
•
•
•
Y, Y_0, Y_1 . . . . Y_9
Z, Z_0, Z_1 . . . . Z_9

Cela représente néanmoins un stock de noms de variables suffisant, mais il n'est pas possible de donner des noms mnémoniques à des variables, ce qui constitue un handicap pour la compréhension des programmes qui ne sont pas bien documentés.

En fait, maintenant sur les BASIC étendus des micro-ordinateurs, le deuxième caractère peut être un caractère alphanumérique, ce qui élargit considérablement le nombre de variables possibles. Il est souvent possible d'utiliser les identificateurs plus longs, mais seuls les deux premiers caractères sont utiles et pris en compte.

Les variables numériques de type entier et réel

En BASIC, il n'y a pas d'instructions de déclarations explicites permettant de définir des variables entières ou réelles, il n'y a pas non plus d'identificateurs, qui sont implicitement considérés comme entiers.

Pour définir une *variable entière* on associe un identificateur à un caractère explicite qui est le caractère %.

Ainsi, l'identificateur N% est considéré comme un nom de variable entière.

Dans tout programme, une telle variable ne pourra contenir que des valeurs entières positives ou négatives.

De telles variables sont représentées de façon interne à la machine par des valeurs binaires exprimées par des chiffres binaires (les bits). Une machine étant caractérisée par des mots machines qui ont une certaine taille (8 bits, 16 bits, 32 bits...), il n'est pas possible de représenter dans un mot machine des nombres entiers supérieurs au nombre maximum que l'on peut représenter en binaire dans ce mot.

Ainsi, si l'on a des mots de 8 bits, on peut représenter les nombres positifs de 0 à $2^8 - 1$, soit 255, ou bien encore les nombres positifs et négatifs de − 128 à + 127.

Pour plus de détails sur ces notions de calcul binaire, nous conseillons au lecteur de se reporter à l'appendice I.

De manière générale, si l'on a des mots de *n* bits, les nombres entiers que l'on peut représenter dans un mot sont tels que :

$$-2^{n-1} \leqslant N \leqslant 2^{n-1} - 1$$

Exemple :

Si le mot machine est de 16 bits on a :

$$-2^{15} \leqslant N \leqslant 2^{15} - 1$$

soit :

$$-32\,768 \leqslant N \leqslant 32\,767$$

Les *variables réelles* sont définies par des identificateurs normaux (N, X..., par exemple). Elles contiennent des valeurs dites réelles qui sont des nombres rationnels comprenant une partie entière et une partie décimale. Là aussi, pour le mode de représentation interne, voir l'annexe I. Le champ de variation pour les variables réelles est bien plus important avec le mode de représentation en virgule flottante (voir ci-dessous). Pour des systèmes BASIC typiques sur micro-ordinateurs on a un champ de variation de 10^{-38} à 10^{38} avec 9 chiffres significatifs.

Remarque. – Les formes de représentation internes des réels et des entiers sont différentes. Pour passer de l'une à l'autre, il faut effectuer une conversion.

Restrictions sur les variables entières

Les variables entières ne sont pas autorisées dans les instructions POUR (FOR) et les instructions de définition de fonctions (DEF) :

- les opérations arithmétiques sont effectuées en réel et les valeurs entières sont donc converties en réels avant tout calcul ;
- les arguments des fonctions mathématiques usuelles (trigonométriques, exponentielle, logarithmique..., etc.) sont également convertis en réel ;
- le passage d'une valeur réelle à une valeur entière implique une opération de tronquage de la partie décimale.

Ainsi, si l'on veut exécuter les instructions :

```
10        N% = 4.556
20        PRINT N%
```

on obtient :

□ 4.

Mais attention, si l'on a :

```
10        N% = -0.5.
20        PRINT N%
```

on obtient :

□ -1

Ou bien, si l'on a :

```
10        N% = 0.98
20        PRINT N%
```

on aura :

□ 0

Dans tous les cas, on obtient donc la partie entière du nombre comme si l'on avait utilisé la fonction partie entière ENT (INT).

L'intérêt des variables entières est dû essentiellement au gain de place en mémoire obtenu par rapport aux nombres réels. Ces variables ont en particulier un intérêt lorsque l'on veut définir des indices de listes ou de tableaux, car ce sont nécessairement des valeurs entières.

Les identificateurs de variables chaînes de caractères

Certains BASIC limitent l'identificateur à une seule lettre, mais, pour la plupart des BASIC, les règles sont les mêmes que celles des identificateurs de variables numériques.

Il faut simplement rajouter le caractère $ après l'identificateur.

Exemple :

A$, B$ .... Z$ .....
A1$ .... Z9$

représente les identificateurs de variables chaînes de caractères.

On a donc là aussi une possibilité de 26 à 286 identificateurs possibles suivant les BASIC.

Les BASIC étendus acceptent tout identificateur commençant par une lettre et suivi d'un caractère alphanumérique.

2-2. Les constantes

Les constantes sont également de deux types :

- les constantes numériques ;
- les constantes chaînes de caractères.

a) *Les constantes numériques*

Ce sont des nombres qui peuvent être entiers ou fractionnaires avec un signe + ou −.

Si le signe est positif il peut être omis. Les chiffres décimaux sont représentés par une partie entière et une partie décimale séparées par *un point* et non par une virgule.

Exemple :

146 est une constante entière.
84 est une constante entière.
153.4 est une constante fractionnaire décimale encore appelée réelle.
3.14 est une constante fractionnaire décimale encore appelée réelle, mais le nombre de décimales est limité.

Remarque. – Lorsque la partie entière est nulle, on peut omettre le 0. Ainsi, la constante 0.5, peut être écrite : .5

La représentation en virgule flottante

De manière interne, les constantes réelles sont codées à l'aide d'une représentation dite en virgule flottante.

Le principe en est simple, puisqu'il est basé sur une représentation exponentielle.

En effet, un nombre tel que : 454.87 peut être représenté sous différentes formes telles que :

$$4.5487 \times 10^{2}$$
$$45487 \times 10^{-2}$$
$$0.45487 \times 10^{3}$$

On voit donc qu'en multipliant par une puissance de 10 positive ou négative on peut faire déplacer le point à l'intérieur de l'ensemble des chiffres significatifs, d'où le nom de virgule (ou point) flottante. Dans cette représentation les chiffres significatifs constituent ce que l'on appelle la *mantisse*. En connaissant la puissance de 10 associée, on peut donc connaître le nombre.

Il existe une forme dite *normalisée* qui est telle que la mantisse soit comprise entre 0.1 et 0.99... Dans l'exemple ci-dessus, 0.45487×10^3 est la forme normalisée. La forme normalisée est utilisée quelquefois en sortie par les interpréteurs BASIC. Dans ce cas, pour représenter un nombre flottant, on utilisera le caractère E comme séparateur entre la mantisse et l'exposant.

Exemples :

0.314 E 1 = 3.14
.14768 E 3 = 147.68
.5 E 0 = 0.5
.25 E − 2 = 0.0025

Remarque. – Cette notation, bien que facile à interpréter pour des utilisateurs scientifiques, n'a pas la faveur des utilisateurs de gestion, car elle est inacceptable pour l'édition de documents tels que facture, relevés bancaires...

Il faut noter que la plupart des BASIC disposent de programmes de conversion d'une forme flottante interne à la forme décimale usuelle.

b) *Les constantes chaînes de caractères*

Nous avons déjà vu des exemples dans le cas où l'on veut imprimer du texte.

La règle concernant ces constantes est qu'elles sont encadrées par des caractères spéciaux : " (guillemets).

Exemple :

" JULES ", " PARIS "
" CECI EST UNE CHAINE "

La seule contrainte est la longueur des chaînes de caractères : elle est fixée à 255 caractères maximum, y compris les caractères blancs. En BASIC, il existe des instructions de manipulation de chaînes de caractères assez complètes qui seront présentées dans un paragraphe ultérieur.

Exercices

1. *Les noms suivants sont-ils corrects en BASIC étendu ?*
   N1, JOJO, PA, R$, LUNE, 1K, B100, 44A, OSS117, M + 1
2. *Les constantes suivantes sont-elles correctes ?*
   15 *24,15* *72.54* 3^2 *−14* ± 31.1 .55
   0.68E + 2 1.24, E − 4 2/3/79.
3. *Ecrire quelques constantes chaînes de caractères. Peut-on utiliser des mots clés à l'intérieur des chaînes de caractères ?*

REPONSES

1. OUI - NON - OUI - OUI - NON - NON - NON- NON - NON - NON -
2. OUI - NON - OUI - NON - OUI - NON - OUI - OUI - NON - NON -
3. " IL SUFFIT DE LES IMAGINER ".

Oui, on peut utiliser des mots clés à l'intérieur des chaînes de caractères.

3. LES MOTS CLES DU LANGAGE

Les mots clés ou mots réservés du langage BASIC sont à l'origine en anglais. Nous les donnerons en français avec la traduction anglaise.

Nous les avons classés par catégorie et dans l'index ils sont donnés par ordre alphabétique.

Mots clés de déclarations

DIM	déclaration de dimension de tableau
DONNEE	DATA

Mots clés conditionnels

SI ALORS	IF THEN

Mots clés itératifs

POUR A PAS	FOR TO STEP
SUIVANT	NEXT

Mots clés impératifs

REM	Remarques
SOIT	LET
ENTRER	INPUT
IMPRIMER	PRINT
DEF FN	Définition de fonction
LIRE	READ
RELIRE	RESTORE

Mots clés de contrôle

SUR	ON
RETOUR	RETURN
ALLER A	GO TO
SOUS PROG	GOSUB

Mots clés opérateurs

Dans certains BASIC existent des opérateurs logiques suivants :

ET	AND
OU	OR
NON	NOT

Mots clés de fonctions mathématiques

FN	Fonction définie par le programmeur
ABS	Valeur absolue
EXP	Exponentielle
LOG	Logarithme népérien
DLOG	Logarithme décimal (sur certains BASIC)
COS	Cosinus
SIN	sinus
TAN	Tangente
ATN	Arc Tangente
RAC (SQR)	Racine carrée
INT	Valeur entière
RND	Générateur aléatoire
SGN	Signe
DEG	Conversion radians degrés (sur certains BASIC)
RAD	Conversion degrés radians (sur certains BASIC)

Mots clés de fonction de manipulation de chaînes de caractères

ASC		Conversion ASCII décimale
CAR$	CHR$	Conversion décimale ASCII
LON	LEN	Longueur d'une chaîne de caractères
VAL		Conversion chaîne de caractères-nombre
CHA$	STR$	Chaîne de caractères correspondant à un nombre
MIL$	MID$	Extraction d'une sous-chaîne de caractères.
DROITE$	RIGHT$	Extraction d'une chaîne à partir de la droite

GAUCHE$	LEFT$	Extraction d'une chaîne à partir de la gauche.

Cette liste de mots clés représente le standard BASIC le plus étendu.

Dans certains BASIC sur micro-ordinateurs on peut également trouver les mots clés.

PEEK	Lecture d'un mot mémoire
POKE	Ecriture d'un mot mémoire
CALL	Appel à un sous-programme en langage
ou USEP	Machine

Il existe d'autre part des mots clés plus spécifiques aux BASIC permettant de faire du graphique.

On trouve également sur certains BASIC évolués des mots clés correspondant à des manipulations de matrices (MAT, ZER, TRN, INV, IDN, CON).

Enfin, il existe également des instructions de manipulation de fichiers sur les BASIC étendus, y compris sur les micro-ordinateurs. Nous y reviendrons dans un chapitre ultérieur.

4. LES REGLES DE PROGRAMMATION DANS LE LANGAGE BASIC

Jusqu'à présent, nous avons donné des exemples et défini les éléments de base du langage : l'alphabet, les identificateurs, les constantes, les mots clés..., etc. Il faut maintenant apprendre à s'en servir et à créer des phrases ou des instructions qui soient correctes syntaxiquement, c'est-à-dire qui doivent respecter les règles de grammaire du langage. En programmation, ces règles sont précises, voire rigides pour le néophyte. Il est donc important de bien connaître ces règles et de bien les utiliser.

La deuxième étape consiste à créer un programme qui ait un sens ! On se situe là à un niveau qui ne dépend plus de la machine, ni du langage, mais de la pensée de l'utilisateur. On se trouve alors au niveau de ce que l'on appelle la sémantique, c'est-à-dire du sens de ce que l'on a programmé.

Les exemples que nous donnerons sont simples et ont été choisis pour leur valeur pédagogique, et pour montrer ce qu'il est possible de faire avec le langage.

Ces exemples doivent permettre à l'utilisateur d'acquérir un certain nombre de réflexes, mais il est bien évident que seule la *pratique* d'un langage permet de mieux le maîtriser.

Il est donc important pour les utilisateurs de ne pas se contenter de lire des programmes ou même de les recopier sur une machine, mais il est nécessaire de prendre des problèmes nouveaux et de les mener à bien tout seul. Dans ce qui suit, nous espérons permettre au lecteur d'appréhender tous les concepts utiles à la programmation et de lui fournir tous les outils correspondant au langage étudié.

Dans la suite de ce chapitre, nous reviendrons en détail sur les différents types d'instructions : les instructions arithmétiques, les instructions de test, les instructions d'itération, les instructions d'entrée-sortie, les instructions de manipulation de chaînes de caractères.

4-1. Les différents types d'instruction en BASIC

Si l'on met à part l'instruction REM, qui ne représente qu'un commentaire ou une remarque on distingue cinq grandes catégories d'instructions :

- les instructions de déclarations (DIM, DATA) ;
- les instructions d'affectation (LET), parmi lesquelles on trouve les instructions arithmétiques et les instructions de manipulation de chaînes ;
- les instructions de test et de rupture de séquence SI... ALORS... ALLER A (IF... THEN... GO TO) ;
- les instructions de boucle d'itération POUR... (FOR...) ;
- les instructions d'entrées-sorties (INPUT, PRINT, READ, WRITE).

Structure d'une instruction

En BASIC standard il est habituel de n'accepter qu'une seule instruction par ligne. Une ligne étant composée de 80 caractères. Une instruction est alors composée d'un numéro suivi du texte de l'instruction : Ce numéro est aussi appelé étiquette.

Exemple :

```
10      LET I = 1
20      INPUT N
```

Cette règle est mise en défaut actuellement, car dans la plupart des BASIC étendus il est possible de définir plusieurs instructions sur la même ligne à condition de séparer chaque instruction par un caractère délimiteur qui est en général représenté par les deux points : .

Dans ce cas il n'est pas possible d'étiqueter ou de numéroter chaque instruction, c'est-à-dire que l'on ne pourra pas y faire référence dans le reste du programme.

Exemple :

```
10        LET I = 1 : INPUT N
```

Ici, par exemple, l'instruction INPUT N ne peut pas être référencée dans le reste du programme. Il faut donc être prudent lorsque l'on définit des lignes avec plusieurs instructions. Si l'on veut d'autre part avoir une compatibilité entre différents BASIC, il vaut mieux ne pas utiliser cette possibilité.

Remarquons cependant que, sur la plupart des micro-ordinateurs actuels, les interpréteurs ont été réalisés par la même société (MICROSOFT) et que par conséquent les comptabilités sont assez bonnes.

Le problème se pose lorsque l'on passe de versions de BASIC étendu à des BASIC standards ou anciens.

A ce titre, il faut remarquer que les BASIC des micro-ordinateurs sont bien souvent aussi sophistiqués que ceux que l'on peut trouver sur des machines plus puissantes.

Pour en finir avec un programme : Instruction FIN et STOP

Avant d'étudier les différents types d'instructions nous allons apprendre comment terminer un programme ! Il y a tout d'abord l'instruction FIN (END) : elle doit donc être la dernière instruction du programme. Lorsque l'interpréteur rencontre cette instruction il arrête l'exécution du programme et il y a retour au moniteur.

Cependant, si l'on veut pouvoir distinguer la fin du programme et l'action d'arrêter l'exécution du programme, qui peut être multiple dans un même programme, on utilisera l'instruction STOP. Cette instruction n'est pas nécessairement la dernière instruction du programme, mais lorsque l'interpréteur la rencontre, il y a arrêt du programme et retour au moniteur. Il peut donc y avoir plusieurs instructions STOP dans un même programme. Il est également possible de relancer un programme après une instruction STOP (commande CONT).

4-2. Instructions de calcul arithmétique

Nous avons vu dans les chapitres précédents des exemples simples d'instructions de calcul arithmétique. En programmation, les instructions arithmétiques permettent de préciser des calculs que l'on a plus souvent l'habitude d'appeler calculs algébriques. En BASIC standard, ces instructions commencent par le mot clé LET (SOIT), suivi d'une instruction arithmétique.

Cependant, sur la plupart des BASIC actuels, le mot clé LET est *facultatif*. Dans ce cas on peut écrire directement l'instruction arithmétique après le numéro d'identification de l'instruction.

Ainsi :

10 LET A = B + C

est équivalent à

10 A = B + C

LES INSTRUCTIONS ARITHMETIQUES OU INSTRUCTIONS D'AFFECTATION

Une telle instruction est caractérisée par un membre gauche qui doit contenir un *identificateur* de variable et par un membre droit contenant une expression arithmétique. Les deux membres sont séparés par un signe = que pour l'instant nous avons considéré comme étant un signe d'égalité. En fait, ce signe a une autre signification en programmation et on l'appelle le plus souvent signe *d'affectation* (dans d'autres langages de programmation, on le note := ou ←.).

En effet, ce signe indique que la machine doit effectuer le calcul de l'expression qui se trouve dans le membre droit de l'instruction et que le résultat obtenu doit être affecté et mémorisé dans la variable qui définit le membre gauche de l'instruction.

Ainsi, en programmation une instruction telle que :

10 X = X + 2

est tout à fait correcte. Elle ne doit donc pas être prise comme une égalité ni une équation. Elle indique que l'on effectue l'addition du contenu de la variable X et de la constante 2 et que le résultat est affecté au membre gauche, qui se trouve également être la variable X. Autrement dit, une telle instruction permet d'ajouter 2 à X, et il est alors évident que l'ancien contenu de X est perdu. Ainsi, par exemple, si le contenu de X était 5 avant l'exécution de l'instruction 10, le résultat terminal sera de 5 + 2, soit 7, et la nouvelle valeur de X sera donc 7 après l'exécution de l'instruction 10.

On peut donc dire que l'instruction d'affectation permet de modifier de façon dynamique le contenu d'une variable. Ceci peut être mis à profit dans des algorithmes de type itératif basés sur des formules de récurrence.

Exemple :

Soit le problème du calcul de la somme des N premiers nombres entiers.

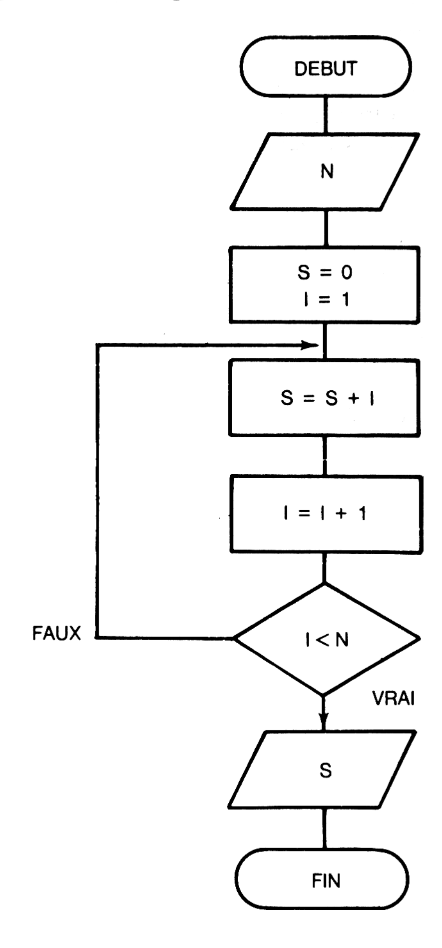
L'algorithme peut être spécifié par une formule de récurrence ; en effet, si l'on suppose connue la somme des $i - 1$ premiers nombres soit S_{i-1}, on obtient la somme des i premiers nombres en rajoutant i à S_{i-1}.

$$\text{Soit : } S_i = S_{i-1} + i$$

L'algorithme est alors le suivant :

- I_1 LIRE N
- I_2 S = 0
- I_3 I = 1
- I_4 S = S + I
- I_5 I = I + 1
- I_6 SI I > N alors imprimer S STOP
- I_7 Sinon continuer en I_4

L'organigramme correspondant est alors :



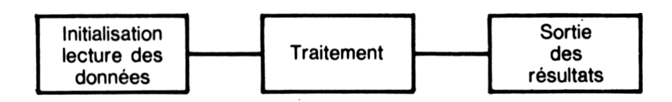
La programmation directe de cet organigramme en BASIQUE et en BASIC donne :

10	ENTRER N	10	INPUT N
20	S = 0	20	S = 0
30	I = 1	30	I = 1
40	S = S + I	40	S = S + I
50	I = I + 1	50	I = I + 1
60	SI I< = N ALORS 40	60	IF I < = N THEN 40
70	IMPRIMER S	70	PRINT S
80	FIN	80	END

En fait, cette façon de programmer ce problème n'est pas la meilleure ni la plus concise, et l'on aurait pu utiliser l'instruction POUR. Nous y reviendrons dans le paragraphe correspondant à cette instruction. Pour l'instant revenons sur cet exemple :

Le programme comprend trois parties :
- une partie statique correspondant à l'initialisation des variables et à la lecture des données (instructions 10, 20 et 30) ;
- une partie dynamique correspondant à l'accumulation des résultats intermédiaires dans les variables S et I (instructions 40, 50, 60) cette partie correspond à la formule de récurrence associée à l'algorithme ;
- une partie terminale qui est également statique, qui imprime les résultats.

Cette structure se retrouve en général dans tout algorithme :
- *Première partie :* initialisation des variables et/ou lecture des données ;
- *Deuxième partie :* elle est dynamique et correspond au traitement à effectuer ;
- *Troisième partie :* Elle est terminale et correspond à la sortie de résultats et à l'arrêt de l'algorithme.



LES DIFFERENTS TYPES D'EXPRESSIONS ARITHMETIQUES

Une instruction arithmétique est de la forme :

SOIT V = Expression arithmétique

où V est une variable et = est le signe d'affectation.

Le mot clé SOIT (LET) est facultatif sur la plupart des BASIC. Dans la suite nous ne l'utiliserons plus.

Une expression arithmétique simple est caractérisée par :

- une suite de noms de variables ou de constantes séparées par des opérateurs arithmétiques.

Les *opérateurs arithmétiques* sont au nombre de cinq :

+ L'addition
− La soustraction
* La multiplication
/ La division
↑ L'élévation à la puissance ou exponentiation.

Pour traduire une expression mathématique quelconque, il suffit de la linéariser en utilisant les opérateurs définis ci-dessus. Il faut cependant prendre en compte toutes les opérations, même si elles ne sont pas explicites dans l'expression initiale.

Exemple :

L'expression $ab + cd$ se traduit par :

A * B + C * D

de même :

$$b^2 - 4\,ac$$

se traduit par :

B↑2 − 4 * A * C

Règles de précédence ou hiérarchie des opérateurs

Considérons l'expression :

A * B / C * D

En l'absence de règles de précédence, elle peut être interprétée de deux manières :

$$a \times b / c \times d \text{ ou } \frac{a \times b}{c} \times d$$

En effet, en BASIC, les expressions ne peuvent faire appel à des artifices typographiques usuels en mathématiques. Il faut donc définir un certain nombre de règles qui définissent une interprétation unique d'une expression donnée. Pour cela on définit une hiérarchie des opérateurs arithmétiques encore appelée *précédence des opérateurs*.

On définit d'abord les règles concernant les expressions arithmétiques *simples*, c'est-à-dire ne faisant pas intervenir de parenthèses ni de fonctions :

Règle 1. - Une expression arithmétique simple est évaluée en tenant compte d'abord des opérations d'exponentiation (↑).

Puis, en allant de la gauche vers la droite, on effectue les opérations de multiplication (*) *ou* de division (/).

Et enfin, toujours en allant de la gauche vers la droite, les opérations d'addition (+) et de soustraction (–).

On a donc le tableau suivant :

– exponentiation ↑	priorité 1
– multiplication * division /	priorité 2
– addition + soustraction –	priorité 3

TABLEAU 1

Exemple

Si l'on reprend l'expression ci-dessus :

A * B / C * D

elle est donc évaluée de la façon suivante :

- tous les opérateurs ont la priorité 2 ;
- on effectue les opérations en allant de la gauche vers la droite, soit :

. $A \times B$

. $\frac{A \times B}{C}$

. $\frac{A \times B}{C} \times D$

L'expression mathématique correspondante est donc :

$$\frac{A \times B \times D}{C} \text{ et non pas } \frac{A \times B}{C \times D}$$

Soit maintenant une expression contenant tous les opérateurs :

A / B + C – D / E * F ↑ 2.

Cette expression sera évaluée en effectuant d'abord l'opération F^2 (F↑2)

puis $\frac{A}{B}$ et $\frac{D}{E} \times F^2$

enfin : $\frac{A}{B} + C - \frac{D}{E} \times F^2$

EXPRESSIONS MATHEMATIQUES COMPLEXES

Introduction de parenthèses

Certaines expressions mathématiques plus complexes nécessitent l'introduction de parenthèses. En effet, en programmation, il n'est pas possible de définir des expressions complexes avec des artifices typographiques utilisant des « échafaudages » tels que :

$$\frac{a + \frac{b}{c}}{c + \frac{d + e}{k}}$$

Pour cela il faut introduire des parenthèses. Ici le numérateur est équivalent à : A + B / C.

Et le dénominateur est équivalent à : C + D / K + E / K.

En effet, l'expression C + D + E / K n'est pas correcte pour représenter $c + \frac{d + e}{k}$. Par contre, si l'on introduit des parenthèses, on peut alors écrire :

C + (D + E) / K

De même, pour indiquer la division de l'expression numérateur par l'expression dénominateur, il faut également introduire des parenthèses. Ainsi l'expression finale correcte est :

(A + B / C) / (C + (D + E) / K)

On en déduit donc la règle suivante :

Règle 2 – Les expressions entre parenthèses sont évaluées avec une priorité supérieure à toutes les autres opérations.

On peut donc considérer que les parenthèses ont la priorité 0.

Remarque. – Certaines expressions peuvent être programmées de plusieurs manières. Ainsi l'expression $\frac{a \times b}{c \times d}$ peut être programmée de trois façons :

– la première forme est :

(A * B) / (C * D)

– la deuxième est :

A * B / (C * D)

Cette expression est en effet correcte puisque l'on effectue d'abord (C × D), puis A × B que l'on divise par C × D.

– la troisième forme est :

A * B / C / D

Dans ce cas, il n'y a pas de parenthèses et cependant le résultat est correct. En effet, on effectue A × B que l'on divise par C, soit : $\frac{A \times B}{C}$, que l'on divise encore par D, soit : $\frac{A \times B}{C \times D}$.

On voit donc que la programmation d'une formule de type fraction ne nécessite pas toujours de parenthèses. On peut cependant remarquer que l'introduction de parenthèses inutiles est préférable à l'écriture d'une forme incorrecte. Ainsi, pour un débutant, il vaut mieux introduire trop de parenthèses de manière à être certain de la validité de l'expression.

Les fonctions mathématiques standard de BASIC

Certaines expressions mathématiques utilisent des fonctions mathématiques standard : racine carrée, exponentielle, logarithme, fonctions trigonométriques..., etc.

Dans ce cas, une fonction est caractérisée par le *nom* de la fonction, qui est un mot clé du langage suivi entre parenthèses d'une expression arithmétique. Ceci suppose évidemment que l'expression entre parenthèses corresponde au domaine de définition de la fonction : ainsi si l'on utilise la fonction racine carrée cela suppose que l'expression associée est toujours positive ou nulle. C'est bien sûr la tâche du programmeur de s'assurer que cela est toujours vérifié, faute de quoi le programme pourra être syntaxiquement correct, mais pourra donner des erreurs à l'exécution. La liste des fonctions mathématiques usuelles en BASIC est donnée sur le tableau 2.

TABLEAU 2 : FONCTIONS MATHEMATIQUES STANDARDS EN BASIC

ABS (X)	Valeur absolue de X
ATN (X)	Arc tangente de X
COS (X)	Cosinus de X
EXP (X)	Exponentielle de X
INT (X)	Partie entière de X
LOG (X)	Logarithme népérien de X
RND (X)	Valeur aléatoire entre 0 et 1
SGN (X)	Signe de X (+ 1 si + , 0 si 0, − 1 si −)
SIN (X)	Sinus de X
SQR (X)	Racine carrée de X (« square root »)
TAN (X)	Tangente de X

Remarque. – En BASIQUE, seules les fonctions INT, RND et SQR ont des mots clés différents :

ENT, ALE et RAC peuvent alors être utilisés.

Dans certains BASIC on peut trouver d'autres fonctions telles que :

- COT (X) pour cotangente X ;
- DLOG (X) Logarithme décimal.

Dans tous les cas cependant les fonctions supplémentaires que l'on peut trouver sont toujours calculables à partir des fonctions standards. Ainsi, par exemple :

COT (X) = 1/TAN (X)
DLOG (X) = LOG (X) / LOG (10)

Il n'est donc pas nécessaire de disposer de ces fonctions pour pouvoir les définir. La liste de ces fonctions dérivées est donnée plus loin.

Introduction de fonctions standards dans une expression arithmétique

Les arguments de fonctions mathématiques peuvent être des expressions arithmétiques entre parenthèses. On retrouve donc à ce niveau la priorité 0 pour l'évaluation des expressions correspondantes, mais de plus, lorsque cette expression a été évaluée, la fonction sera simultanément calculée.

On peut donc énoncer la règle suivante :

Règle 3. – Dans une expression arithmétique les fonctions mathématiques sont évaluées avec la priorité 0 équivalente aux expressions entre parenthèses.

Exemples :

Soit l'expression :

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Elle est programmée sous la forme :

(– B + SQR (B ↑ 2 – 4 * A * C)) / (2 * A)

On effectue d'abord les expressions entre parenthèses et les fonctions, soit :

SQR (B ↑ 2 – 4 * A * C)

puis :

– B + SQR (B ↑ 2 – 4 * A * C)

puis :

2 * A

et enfin on divise la première expression par la deuxième.

L'opérateur unaire négation

Soit l'expression :

$$R = C \times \frac{I}{1 - (1 + I)^{-N}}$$

qui représente le calcul des remboursements R d'un capital C remboursé sur N années, avec un taux d'intérêt de I %

Cette expression peut être programmée par :

R = C * I / (1 - (1 + I) ↑ (- N))

En BASIC étendu, on peut également l'écrire :

R = C * I / (1 - (1 + I)↑ - N)

On voit dans cette expression l'introduction du signe - pour représenter l'opération de négation (ici - N).

Dans ce cas, on peut considérer l'opérateur négation qui est aussi représenté par le symbole - comme un opérateur individuel n'agissant que sur un opérande (on dit aussi *unaire* ou *monadique*). Cet opérateur aura une priorité supérieure aux opérations faisant intervenir deux opérandes (on dit aussi diadique) tels que * / ou + -. Ainsi l'opérateur - monadique peut être considéré comme ayant une priorité 1*bis* entre l'exponentiation et la multiplication division.

Exemple :

L'expression - A * B + C est équivalente à :

(- A) * B + C

ce qui revient à effectuer d'abord l'opération de négation sur A avant de multiplier par B et enfin ajouter la valeur C.

RESUME SUR LES INSTRUCTIONS ARITHMETIQUES

Nous avons vu qu'une instruction arithmétique est une instruction qui commence par le mot clé LET (en général facultatif) suivi d'une instruction du type :

(LET) nom de variable = expression arithmétique

Les règles syntaxiques concernant les expressions arithmétiques sont résumées ci-dessous :

Règle 1. - La priorité des opérateurs dans l'évaluation des expressions arithmétiques est :

1. - Parenthèses ou fonctions mathématiques. Les parenthèses et les fonctions les plus internes étant effectuées d'abord.
2. - L'exponentiation (↑).
3. - La négation (-) opération monadique.

4. – La multiplication division (* /) dans l'ordre de leur rencontre en allant de la gauche vers la droite.
5. – L'addition soustraction (+ –) dans l'ordre de rencontre en lisant l'expression de la gauche vers la droite.

Si maintenant de telles expressions sont utilisées dans un programme, il faut également obéir à un certain nombre de règles dites sémantiques, dont les deux principales concernent la connaissance préalable des valeurs de variables et l'homogénéité des expressions.

Règle 2. – Toutes les variables utilisées dans une instruction arithmétique doivent avoir une valeur au moment de l'exécution de l'instruction.

* En général, la valeur prise par défaut est zéro.

Règle 3. – Toutes les variables ou constantes d'une instruction arithmétique doivent être des variables numériques, à l'exclusion de variables ou constantes de type chaînes de caractères.

Exercices sur les instructions arithmétiques

1. *Traduire en BASIC les expressions suivantes :*

$$4x + 5y, \frac{x + y}{z}, (a + b)^2, e^{-(x + y)}, \frac{n(n + 1)}{2}$$

$$\text{Log} \frac{a + b}{c + d}, \sqrt{a^2 + b^2}, \pi r^2, \cos \varphi$$

2. *Etant donné les expressions BASIC suivantes, écrire les expressions mathématiques correspondantes et évaluer ces expressions pour des valeurs de : A = 2, B = 3, C = 4 :*

A * B + C/A (A/2 + B/3 + C/4)/3

A ↑ 2 + B ↑ 2 + 2 * A * B (A + B) ↑ 2

A + SQR (B^2 + C^2)/5 * A C * (1 + B/100) ↑A

3. *Soit la suite d'instructions arithmétiques suivantes : calculer la valeur des trois variables A, B, C après l'exécution de chacune des instructions pour A = 2, B = 3, C = 4.*

```
C = A + B + C/A
B = B + C
A = A * B + C * A ↑ 3
C = A - B * C
B = (A - 4) /C
A = A - B * C
C = B - A
B = B/2
```

SOLUTION DES EXERCICES

1. 4 * X + 5 * Y
   N * (N + 1) / 2
   EXP (− (X + Y))
   SQR (A↑2 + B↑2)
   COS (FI)
   (X + Y) / Z
   (A + B)↑2
   LOG ((A + B) / (C + D))
   PI * R↑2
   (ou π * R↑2 sur certains micro-ordinateurs disposant du caractère π sur le clavier).

2. $ab + \frac{c}{a}$ valeur : 8

   $\frac{\frac{a}{2} + \frac{b}{3} + \frac{c}{4}}{3}$ valeur : 1

   $a^2 + b^2 + 2ab$ valeur : 25

   $(a + b)^2$ valeur : 25

   $a + \frac{\sqrt{b^2 + c^2}}{5} \times a$ valeur : 4

   $c\,(1 + \frac{b}{100})^a$ valeur : 4.2436

3.

	A	B	C
	2	3	4
$c = a + b + c/a$	2	3	7
$b = b + c$	2	10	7
$a = a * b + c * (a)^3$	76	10	7
$c = a - b * c$	76	10	6
$b = \frac{a - 4}{c}$	76	12	6
$a = a - bc$	4	12	6
$c = b - a$	4	12	8
$b = b/2$	4	6	8

Remarque. - Pour le lecteur qui dispose d'une machine travaillant en BASIC, il suffit de rentrer les instructions précédentes et d'insérer au début une instruction INPUT A, B, C, et après chaque instruction une instruction PRINT A, B, C.

Exercices

Convertir en BASIC les expressions suivantes :

a) $2i + 4j$
b) $2a + 3b$
c) $a^2 - 4b$
d) $2(a + b)$
e) $(a + 2b + 3c)^2$
f) $(\frac{a}{b})^2$
g) $\frac{a + b}{c + d}$
h) $\frac{a}{b} + \frac{c}{d}$
i) $a^2 - 4ab$
j) $\frac{x}{2!} + \frac{x^2}{4!}$

4-3. Les instructions de manipulation de chaînes de caractères

Comme pour le calcul arithmétique il existe ce que l'on appelle des constantes chaînes de caractères et des variables de type chaînes de caractères.

Les constantes chaînes de caractères :

Nous avons vu que ces constantes sont déterminées par deux délimiteurs identiques (un au début et un à la fin) qui sont les caractères guillemets : ". La constante est représentée par l'ensemble de caractères compris entre les guillemets.

Exemples

– " BONJOUR " est une constante chaîne de caractères représentant le message BONJOUR.
– " CECI EST UNE LONGUE CHAINE DE CARACTERES " est une chaîne de caractères comprenant plusieurs mots séparés par des blancs.
– " 14 JUILLET 1789 " est aussi une chaîne de caractères indépendamment du fait qu'elle contient des chiffres. Dans ce cas, ils seront pris comme des caractères.

IMPRESSION DIRECTE DE CHAINES DE CARACTERES

De la même façon qu'il est possible d'utiliser le BASIC pour faire des calculs directs, il est possible de l'utiliser pour faire des impressions de chaînes de caractères.

Ainsi, si l'on écrit :

PRINT " BONJOUR "®

La réponse est :

BONJOUR

Ceci présente bien sûr un intérêt limité dans la mesure où il n'y a eu aucun traitement de la chaîne de caractères. Il faut cependant remarquer que si l'on dispose d'une imprimante, on peut ainsi éditer une liste de messages pour un destinataire X sans avoir à savoir programmer.

L'OPERATION DE CONCATENATION

Il s'agit d'une opération très importante, très simple et nécessaire à la manipulation des chaînes de caractères.

Définition. – La concaténation de deux chaînes de caractères est l'opération qui consiste à fusionner les deux chaînes initiales en une seule chaîne obtenue en juxtaposant les caractères de la deuxième chaîne immédiatement après ceux de la première.

Exemple :

– La concaténation de " AUJOURD " et " ' HUI " donne " AUJOURD'HUI ".

– La concaténation de " BONNE " et " NUIT " donne la chaîne " BONNE NUIT ".

– La concaténation du préfixe " ANTI ", de " CONSTITUTION " et de la terminaison " NELLEMENT " donne la chaîne : " ANTICONSTITUTIONNELLEMENT ".

Ainsi, la concaténation est une opération qui permet d'obtenir des chaînes de caractères pour constituer des mots à partir de lettres ou des phrases à partir de mots. C'est donc une opération qui est à la base de tout langage.

Notation de la concaténation en BASIC

Comme il s'agit d'une opération d'ajout, l'opérateur de concaténation est notée + en BASIC.

Il n'y a pas d'ambiguïté entre le signe + arithmétique, dans la mesure où l'on ne peut pas avoir mélange des types de variables numériques et chaînes de caractères.

Exemple :

– " JEAN " + " PIERRE " donne " JEAN PIERRE ".

L'on peut également faire exécuter des ordres d'impression directs en utilisant l'opération de concaténation.

Exemple :

– PRINT "JULES" + "ET" + "JIM" ®

□ JULES ET JIM

– PRINT "BONJOUR" + "MADAME" ®

□ BONJOUR MADAME.

LA REPRESENTATION DES CARACTERES EN MACHINE

On a vu dans le chapitre I que de façon interne une machine manipule les informations sous forme de *mots* machine qui ne peuvent contenir que des informations binaires (suite de 0 ou de 1). Lorsque la machine traite des caractères, il en est de même. Ceci suppose donc que tout caractère soit représenté par un code binaire interne.

Les codes actuellement les plus utilisés sont des codes sur *huit bits* (ce qui permet de représenter jusqu'à 2^8 soit 256 caractères différents).

Le code le plus répandu sur les petites machines et en particulier sur tous les micro-ordinateurs est le code ASCII (« American Standard Communication Informations Interchange »), dont la table de correspondance est donnée sur le tableau suivant.

TABLE DES CODES ASCII

CARACTERE OU CONTROLE	ASCII (HEXA DECIMAL)	CARACTERES SPECIAUX ET CHIFFRES	ASCII (HEXA DECIMAL)	CARACTERES MAJUSCULES	ASCII (HEXA-DECIMAL)	CARACTERES MINUSCULES	ASCII (HEXA-DECIMAL)
NUL	00	SP (espace)	20	@	40	\	60
SOH	01	!	21	A	41	a	61
STX	02	"	22	B	42	b	62
ETX	03	#	23	C	43	c	63
EOT	04	$	24	D	44	d	64
ENQ	05	%	25	E	45	e	65
ACK	06	&	26	F	46	f	66
BEL	07	/	27	G	47	g	67
BS	08	(	28	H	48	h	68
HT	09	)	29	I	49	i	69
LF	0A	*	2A	J	4A	j	6A
VT	0B	+	2B	K	4B	k	6B
FF	0C	,	2C	L	4C	l	6C
CR	0D	−	2D	M	4D	m	6D
SO	0E	.	2E	N	4E	n	6E
SI	0F	/	2F	O	4F	o	6F
DLE	10	0	30	P	50	p	70
DCA (X-ON)	11	1	31	Q	51	q	71
DC2 (TAPE)	12	2	32	R	52	r	72
DC3 (X-OFF)	13	3	33	S	53	s	73
DC4 (TAPE)	14	4	34	T	54	t	74
NAK	15	5	35	U	55	u	75
SYN	16	6	36	V	56	v	76
ETB	17	7	37	W	57	w	77
CAN	18	8	38	X	58	x	78
EM	19	9	39	Y	59	y	79
SUB	1A	:	3A	Z	5A	z	7A
ESC	1B	;	3B	[	5B	{	7B
FS	1C	<	3C	\	5C	|	7C
GS	1D	=	3D	]	5D	}(ALT MODE)	7D
RS	1E	>	3E	Δ(↑)	5E	´	7E
US	1F	?	3F	− (←)	5F	DEL	7F

Il est intéressant de noter sur le tableau des codes ASCII que, si l'on regarde les codes associés aux lettres, on s'aperçoit que ce sont des nombres en ordre croissant lorsque l'on va de A à Z. Par contre, le caractère « blanc » possède un code qui est inférieur à l'ensemble de lettres de l'alphabet. Ces remarques font que lorsque l'on voudra classer des chaînes de caractères par ordre alphabétique, par exemple, on pourra très facilement le faire sur des chaînes de caractères alphabétiques, mais il faudra se méfier des blancs qui peuvent être intercalés. Nous reviendrons sur ce problème un peu plus loin.

Les variables chaînes de caractères

Nous avons vu en début de chapitre que les noms de ces variables étaient composés d'une lettre suivie ou non d'un caractère alphanumérique et se terminent par le caractère $.

Expression simple de chaînes de caractères

Une expression simple de chaîne de caractères est une expression comprenant la concaténation d'une ou de plusieurs variables ou constantes chaînes de caractères.

Exemple : – "LE" + N$
– A$ + B$ + "CHAINE 3"
– NO$ + PR$ + AD$

Instructions d'affectation de chaînes de caractères

Une instruction d'affectation a la même structure que pour les instructions arithmétiques :

Nom de variable chaîne = expression chaîne

Là aussi l'expression du membre droit est d'abord traitée, et le résultat est affecté à la variable située dans le membre gauche. On peut donc là aussi avoir apparition du même nom de variable à gauche et à droite du signe d'affectation (=).

Exemple :

Soit NO$ et PR$ deux variables représentant les chaînes de caractères noms et prénoms.

On peut définir une variable ID$ telle que

```
10      (LET) ID$ = NO$ + PR$
```

On aurait pu également définir :

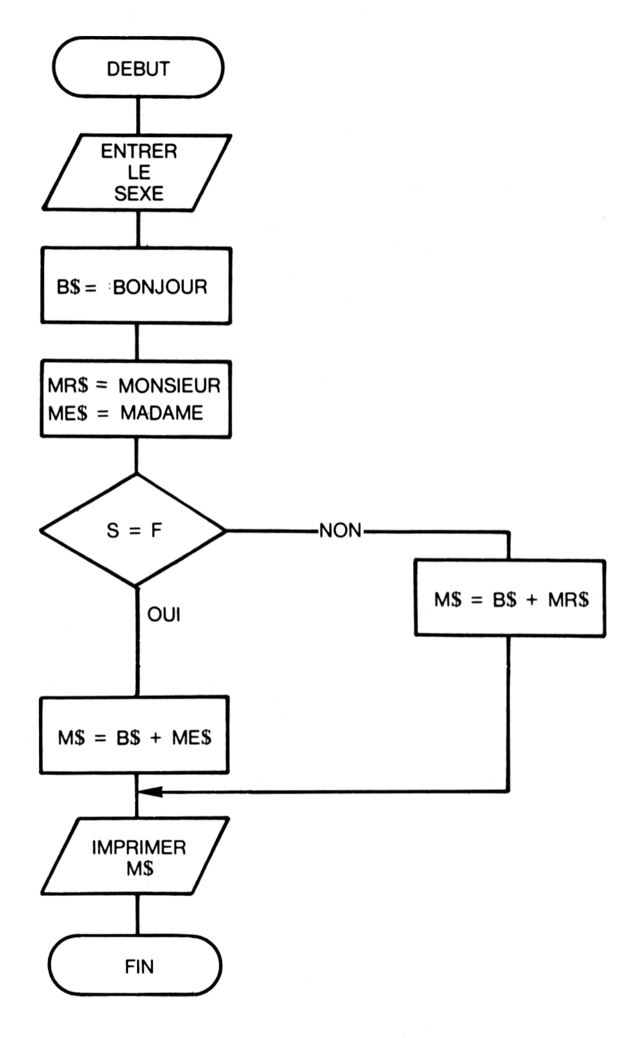
```
10      (LET) NO$ = NO$ + PR$
```

dans ce cas la variable NO$ contiendra à la fin de l'exécution la concaténation des chaînes composant les noms et prénoms.

Traitement des chaînes de caractères

Afin d'éclairer un peu l'utilité de ce qui précède nous allons étudier un programme qui permet d'imprimer « BONJOUR MADAME » ou « BONJOUR MONSIEUR » suivant que l'on répond M ou F à une question relative à son sexe.

L'organigramme est très simple :



on obtient le programme suivant :

```
10      ENTRER S$
20      B$ = " BONJOUR "
```

```
30      ME$ = "MADAME"
40      MR$ = "MONSIEUR"
50      SI S$ = "F" ALORS M$ = B$ + " " + ME$ :
        ALLER A 70
60      M$ = B$ + " " + MR$
70      IMPRIMER M$
80      FIN
```

Soit en BASIC :

```
10      INPUT S$
20 à 40 Identiques
50      IF S$ = "F" THEN M$ = B$ + " " + ME$: GO
        TO 70
60      Identique
70      PRINT M$
80      END
```

Ainsi, à l'exécution si l'on frappe
? F® ou ? M®
l'on obtient :
□ BONJOUR MADAME □ BONJOUR MONSIEUR

Remarque. – Dans les instructions 50 et 60 on a rajouté le caractère blanc représenté par la constante " " pour obtenir une séparation entre les deux mots.

Les fonctions de traitements de chaînes de caractères

De la même façon qu'il existe des fonctions mathématiques, il existe un certain nombre de fonctions relatives aux chaînes de caractères.

1. LA FONCTION LEN

Cette fonction, qui peut être appelée LON en BASIQUE, permet de donner la longueur en nombre de caractères d'une chaîne.

Exemple :

Si l'on écrit :
- PRINT LEN ("TOTO") ®, on obtient
□ 4
- PRINT LEN ("ANTICONSTITUTIONNELLEMENT")
□ 25

2. LES FONCTIONS EXTRACTIONS DE CARACTERES

L'opération de concaténation permet d'ajouter des caractères à une chaîne. Il est donc nécessaire de disposer de fonctions permettant d'extraire une sous-chaîne de caractères à partir d'une

chaîne existante. Pour cela il existe trois fonctions qui sont disponibles sur les BASIC étendus : ce sont les fonctions :

RIGHT$, LEFT$ et MID$.

a) *La fonction d'extraction à DROITE (RIGHT)*

Elle fonctionne avec deux paramètres :

RIGHT$ (X$, K)

Le premier paramètre est une chaîne de caractères (constante ou variable).

Le deuxième est un nombre entier (constante ou variable).

Cette fonction permet d'extraire les K caractères les plus à droite de la chaîne X $.

Exemple :

- PRINT RIGHT$ ("JEAN CLAUDE", 6) ®
  □ CLAUDE

b) *fonction d'extraction à GAUCHE (LEFT$)*

Cette fonction est similaire à la précédente avec LEFT$ (X$, K).

Le premier paramètre est une chaîne de caractères et le deuxième est un nombre entier.

Elle permet d'extraire les K caractères les plus à gauche de la chaîne X$.

Exemple :

- PRINT LEFT$ ("JEAN CLAUDE", 4) ®
  □ JEAN

c) *La fonction d'extraction au MILIEU (MID$)*

Cette fonction permet d'extraire une chaîne de caractères à l'intérieur d'une autre chaîne. Elle comporte trois paramètres : MID$ (X$, K, L).

X$ est une chaîne de caractères (constante ou variable).

K est un entier qui indique à partir de quel caractère doit commencer l'extraction.

L indique le nombre de caractères à extraire.

Exemple :

- PRINT MID$ ("LE CIEL EST BLEU", 8, 3) ®
  □ EST

Dans cet exemple, on a extrait les trois caractères à partir du huitième caractère. On voit que la fonction MID$ est la plus générale, car elle recouvre les fonctions RIGHT$ et LEFT$.

Un exemple de programme simple de traitement de texte

Il s'agit d'un programme permettant d'écrire un mot en sens inverse (de la droite vers la gauche)

```
10      ENTRER A$
20      K =  LON (A$)
30      B$ = " "
40      POUR I =  1 A K
50      A$ = GAUCHE$ (A$, K - I + 1)
60      L$ = DROITE$ (A$, 1)
70      B$ = B$ + L$
80      SUIVANT I
90      IMPRIMER B$
```

Explication. – A$ contient le mot ou la phrase à inverser. K contient la longueur de la chaîne de caractères associée. B$ est initialisé avec une chaîne vide.

Dans la boucle POUR, qui va jusqu'en 80, on réduit la chaîne initiale de 1 à chaque itération et on récupère le dernier caractère à droite pour reconstituer le mot inversé dans B$.

En BASIC, on obtient :

```
 10     INPUT A$
 20     K = LEN (A$)
 30     B$ = " "
 40     FOR I = 1 TO K
 50     A$ = LEFT$ (A$, K - I + 1)
 60     L$ = RIGHT$ (A$, 1)
 70     B$ = B$ + L$
 80     NEXT I
 90     PRINT B$
100     END
```

En faisant une exécution :

```
?       BOULEVARD®
□       DRAVELUOB
```

Exercices sur le traitement de chaînes

1. *Ecrire un programme BASIC qui permette de rechercher si un caractère est présent dans une chaîne de caractères.*

2. *Ecrire un programme qui calcule la fréquence d'apparition d'un caractère dans une chaîne.*
3. *Ecrire un programme qui, à partir de l'exemple donné dans le paragraphe précédent, détermine si un mot ou une phrase est un palindrome, c'est-à-dire, une chaîne de caractères qui est identique dans les deux sens.*

CORRIGES DES EXERCICES

1.
```
 10   INPUT A$, L$
 20   LG = LEN (A$)
 30   FOR I = 1 TO LG
 40   C$ = MID$ (A$, I, 1)
 50   IF C$ = L$ THEN 90
 60   NEXT J
 70   PRINT "LE CARACTERE" ; L$ ; "EST ABSENT"
 80   GO TO 100
 90   PRINT "LE CARACTERE" ; L$ ; "EST PRESENT"
100   END
```

Autre solution

```
1         M$ = "EST ABSENT"
10 à 40   Identique
50        IF C$ = L$ THEN M$ = "EST PRESENT"
60        NEXT I
70        PRINT "LE CARACTERE" ; L$ ; M$ ; "DANS" ; A$
80        END
```

2. La solution du deuxième problème consiste à partir de la solution 1 à compter le nombre de caractères à chaque fois que le test C$ = L$ est vrai : on a donc :

```
10   N = 0
20   INPUT A$, L$
30   LG = LEN (A$)
40   FOR I = 1 TO LG
50   C$ = MID (A$, I, 1)
60   IF C$ = L$ THEN N = N + 1
70   NEXT I
80   PRINT "LE CARACTERE" ; L$ ; "APPARAIT" ;
     N ; "FOIS DANS" ; A$
90   END
```

Exécution

? QUEL BEAU TEMPS, U ®
□ LE CARACTERE U APPARAIT 2 FOIS DANS QUEL BEAU TEMPS.

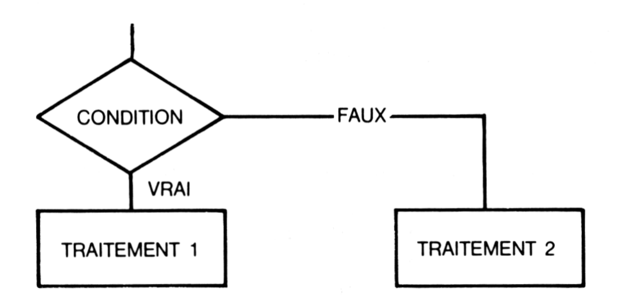
3. Il suffit pour cela de reprendre le programme de renversement d'un mot du paragraphe précédent et de rajouter les instructions :

```
 15    P$ = A$
 85    IF P$ = B$ THEN 110
 90    PRINT P$; "N'EST PAS UN PALINDROME"
100    GO TO 120
110    PRINT P$; "EST UN PALINDROME"
120    END
```

4-4. Les instructions de test ou instructions conditionnelles

Ce sont ces instructions qui donnent véritablement la possibilité de traduire des algorithmes dans un langage de programmation. En effet, suivant la valeur de certaines données ou variables, il va être possible de programmer le test d'une condition relative à ces variables et de prévoir des traitements différents suivant que la condition est vérifiée ou non.

La représentation d'un test est donnée par le schéma d'organigramme suivant :



Elle peut être interprétée en langage clair par :

Si la condition est vraie ALORS effectuer le traitement 1 SINON effectuer le traitement 2.

On voit sur ce schéma que la sortie d'un test est composée de deux branches sur l'organigramme. Comme un programme est écrit sous forme d'instructions en séquence, cette structure implique une instruction du type SI... ALORS... SINON..., chacun des mots clés étant séparé par des instructions correspondant aux deux termes de l'alternative. Cette solution n'est pas utilisée en BASIC standard, bien qu'elle ait été proposée sur certains interpréteurs. L'autre façon

de programmer une telle structure est de se limiter à une instruction de type :

SI. . . . . .ALORS. . . . . .

L'autre terme de l'alternative (le SINON) étant représenté par la suite du programme. Dans ce cas, la séquence d'instructions qui suit l'instruction conditionnelle représente le traitement à effectuer lorsque la condition n'est pas vérifiée.

Instruction conditionnelle en BASIC

En BASIC, c'est la deuxième solution qui est la plus répandue et constitue le standard. On a donc une instruction du type :

SI condition ALORS instruction 1
instruction suite.

Les expressions conditionnelles

La condition qui est testée est exprimée sous forme d'expression relationnelle ou conditionnelle dont il faut définir la structure.

Définition. – Une expression conditionnelle simple est une expression composée de deux expressions de même type (arithmétiques ou chaîne de caractères) séparées par un opérateur relationnel. Une expression conditionnelle est VRAIE ou FAUSSE.

Opérateurs relationnels

Les opérateurs relationnels en BASIC sont au nombre de 6 :

=	égalité
< >	différent
>	supérieur
<	inférieur
> =	supérieur ou égal
< =	inférieur ou égal.

Table des opérateurs relationnels

Exemples :

- A< >B — A différent de B
- C> = 10 — C supérieur ou égal à 10
- A<B + C — A inférieur à B + C

De telles expressions sont courantes en mathématiques. En BASIC, elles sont également utilisables pour les chaînes de caractères. L'égalité = veut dire identité entre chaînes.

Les opérateurs < et > utilisés pour comparer des chaînes de caractères indiquent que les codes associés sont inférieurs ou

supérieurs. De façon pratique, comme l'on a vu que les codes des lettres étaient des nombres croissants, cela revient à pouvoir comparer des chaînes de lettres suivant l'ordre alphabétique.

Exemple :

Ainsi, l'expression A$ < B$ permet de définir la condition : est-ce que la chaîne A$ est classée alphabétiquement avant la chaîne B$? De même A$ = B$ représente l'identité de deux chaînes.

Remarque. – On peut donc supposer que des algorithmes de tri développés sur des valeurs numériques seront valables pour des valeurs alphabétiques.

Il est également possible de définir des expressions relationnelles telles que :

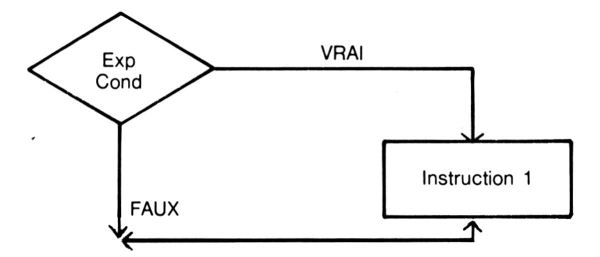
A$ < B$ + C$

Attention. – Il n'est par contre pas possible de mélanger les types de variables ou constantes utilisées dans une même expression.

Instruction conditionnelle simple

Si l'on considère la structure de l'instruction BASIQUE : SI expression conditionnelle ALORS instruction 1.

Elle s'interprète de la façon suivante :



Si l'expression conditionnelle est VRAIE alors on exécute l'instruction 1. SI l'expression conditionnelle est FAUSSE alors on n'exécute pas l'instruction 1, mais on passe à l'instruction suivante.

Exemple :

Soit un nombre réel X ; avant de prendre la racine carrée de sa valeur absolue, il faut s'assurer qu'il est positif, sinon il faut prendre l'opposé de X.

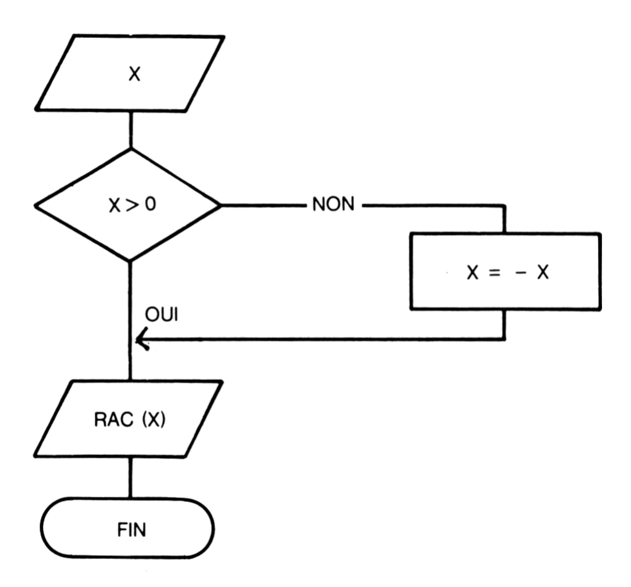
Ceci peut-être programmé par les instructions suivantes :

```
10      INPUT X
20      IF X< 0 THEN X = -X
30      PRINT SQR (X)
40      END
```

Ou en BASIQUE :

```
10      ENTRER X
20      SI X < 0 ALORS X = - X
30      IMPRIMER RAC (X)
40      FIN
```

L'organigramme correspondant est :

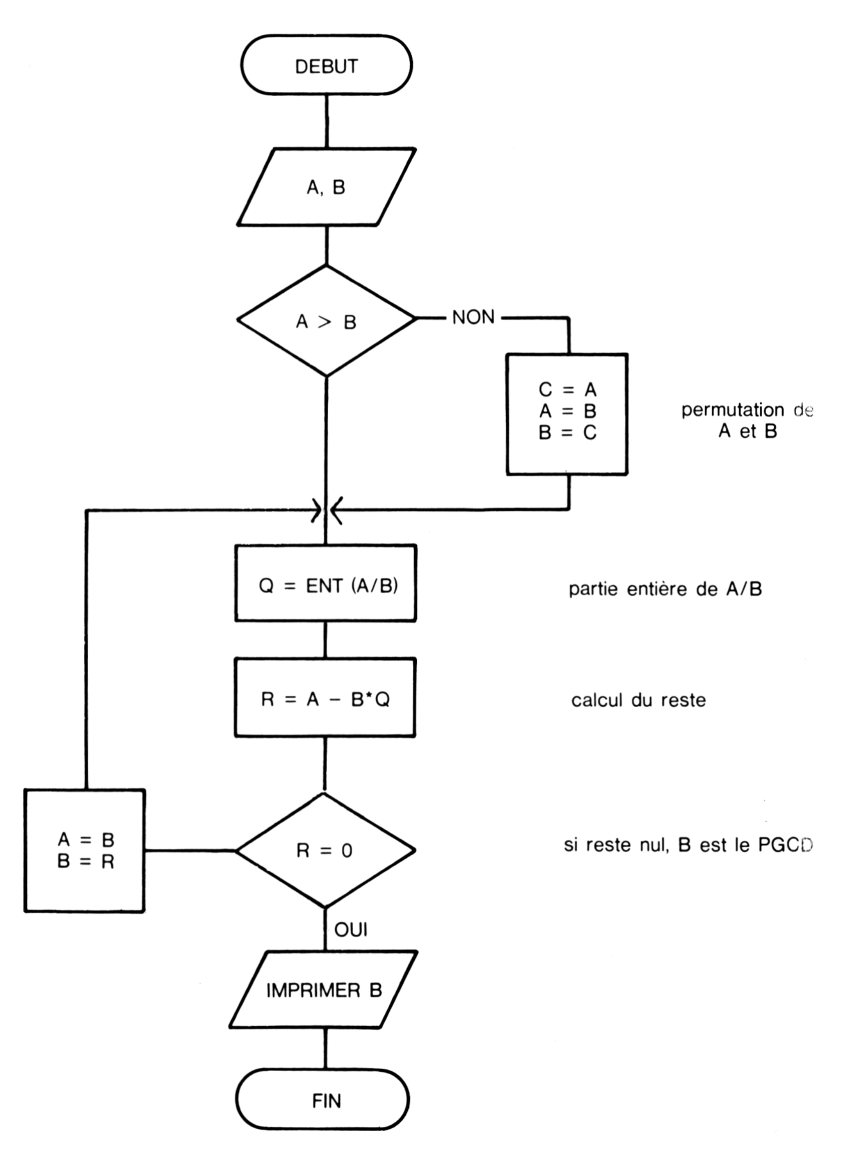


Si l'on a par contre un problème où, à la suite du test, la condition vraie implique un traitement nécessitant plus d'une instruction, on ne peut utiliser cette instruction sans faire appel à un branchement vers une autre instruction du programme.

Exemple : Algorithme d'Euclide.

Soit l'organigramme de l'algorithme d'Euclide déjà donné dans le chapitre II. Il s'agit de calculer le PGCD de deux nombres A et B.

Dans cet organigramme il y a deux tests. Le premier sert à déterminer le plus grand nombre de A et B puis à les échanger. On peut d'ailleurs remarquer ici la nécessité d'utiliser une troisième



variable C pour effectuer cette permutation. Le deuxième test sert à savoir si le reste obtenu est nul, ce qui indique que B est le PGCD.

Dans chacun de ces tests la condition négative ne peut s'exprimer en une seule instruction, comme dans l'exemple précédent. Il faut donc avoir recours à un branchement vers une autre instruction.

Dans une expression conditionnelle un tel branchement s'effectue en précisant le *numéro* de l'instruction correspondante

après le mot clé ALORS (THEN). On peut également le faire précéder du mot clé ALLER A (GO TO).

Ainsi, le programme correspondant s'écrit :

```
 10    INPUT A, B              ENTRER A, B
 20    IF A > B THEN 60        SI A > B ALORS 60
 30    C = A
 40    A = B
 50    B = C
 60    Q = INT (A/B)
 70    R = A - B * Q
 80    IF R = 0 THEN 120       SI R = 0 ALORS 120
 90    A = B
100    B = R
110    GO TO 60                ALLER EN 60
120    PRINT B                 IMPRIMER B
130    END
```

PROGRAMME DE L'ALGORITHME D'EUCLIDE EN BASIC STANDARD

L'instruction de branchement inconditionnel

Dans cet exemple l'instruction ALLER EN ou VA EN (GO TO) est une instruction de branchement inconditionnel qui indique le numéro de l'instruction vers laquelle doit se faire le branchement. On aurait pu également utiliser cette instruction après le THEN (ALORS).

Ainsi IF A > B THEN GO TO 60 est équivalent à IF A > B THEN 60.

L'instruction de branchement inconditionnel est une instruction nécessaire en BASIC, mais il ne faut pas en abuser, car cela peut vite conduire à ce que l'on appelle les programmes « spaghettis », où il y a tellement de branchements que l'on ne s'y retrouve plus. Nous ne reviendrons pas ici sur les avantages et les mérites de la programmation structurée, mais avec la plupart des BASIC étendus disposant d'instructions multiples sur une ligne il est possible d'éviter un grand nombre de branchements.

En reprenant l'exemple précédent on obtient :

```
10    INPUT A, B
20    IF A > B THEN C = A : A = B : B = C
30    Q = INT (A/B)
40    R = A - B * Q
50    IF R < > 0 THEN A = B : B = R : GO TO 30
60    PRINT B
70    END
```

PROGRAMME ALGORITHME D'EUCLIDE EN BASIC ETENDU

On peut remarquer que ce programme est lisible séquentiellement, qu'il est plus compact, ne comporte plus qu'un seul branchement, et est beaucoup plus vite compris que la version précédente. Son seul inconvénient est de ne pas être absolument standard. Ce programme a été testé sur le BASIC du CBM de Commodore et du système APPLE.

Les expressions conditionnelles complexes

Jusqu'à présent nous avons vu des expressions conditionnelles simples, ne faisant intervenir qu'une seule condition. En pratique, on peut rencontrer des algorithmes où l'on ait besoin de conditions complexes du type :

- Si la condition 1 *et* la condition 2 sont vraies, alors...
- Si la condition 1 *ou* la condition 2 est vraie alors...
- Si la condition 1 et pas la condition 2 sont vérifiées alors...

De telles expressions sont appelées des expressions logiques ou booléennes (du nom du mathématicien BOOLE, qui est à l'origine d'une algèbre basée sur des variables ne pouvant prendre que deux valeurs : vrai ou faux). Le BASIC étendu permet de programmer directement de telles expressions.

Les opérations logiques

Il faut tout d'abord définir ce que l'on appelle valeur et variable logiques (ou booléennes).

Une valeur logique ou booléenne est une valeur qui ne peut prendre que deux états : le VRAI (V) ou le FAUX (F). On peut également représenter ces valeurs par 0 et 1. Une variable logique est une variable qui ne peut prendre que des valeurs logiques. Une condition peut donc être représentée par une variable logique.

En BASIC standard il n'est pas possible de définir des variables logiques ni d'utiliser les opérations logiques.

Ce qui suit n'est donc valable que pour les BASIC étendus.

L'opérateur négation logique NON (NOT)

Si l'on considère une variable logique A

- NON A est une variable logique qui est vraie quand A est fausse et inversement.

On peut le représenter par le schéma suivant appelé table de vérité :

A	NON A
Vrai	Faux
Faux	Vrai

Exemple : Si A représente la condition : B < C, NON A représentera la condition inverse B ⩾ C

L'opérateur ET logique (AND)

Dans ce cas, si l'on considère 2 variables logiques A et B, l'expression logique A ET B n'est vraie que si A et B sont vrais *simultanément*. Ceci peut également se représenter par un tableau de vérité.

A \ B	Vrai	Faux
Vrai	Vrai	Faux
Faux	Faux	Faux

A ET B

Exemple :

– Si A est une condition C 1 et B une condition C 2 la condition : C 1 ET C 2 n'est vraie que si C 1 et C 2 sont vraies. En particulier C 1 ET C 2 est fausse lorsque C 1 ou C 2 sont faux.

L'opération OU logique (OR)

L'expression A OU B est vraie si au moins l'une des deux variables logiques A ou B est vraie. On a le tableau de vérité :

A \ B	Vrai	Faux
Vrai	Vrai	Vrai
Faux	Vrai	Faux

A OU B

Exemple :

– Si A et B représentent des conditions C 1 et C 2 la condition C 1 OU C 2 ne sera fausse que si C 1 et C 2 le sont simultanément.

Utilisation des opérateurs logiques en BASIC

Il n'y a pas à proprement parler de variables logiques en BASIC. Cependant les opérateurs logiques sont utilisables pour définir des conditions multiples d'une instruction conditionnelle.

Exemples :

– Soit à tester la valeur d'un nombre qui doit être compris entre une borne inférieure I et une borne supérieure S.

IF N > I AND N < S THEN...
SI N > I ET N < S ALORS...

– Soit à tester une chaîne de caractères pour vérifier que l'on a bien une réponse conforme à la question. Si l'on veut tester que les caractères rentrés correspondent au sexe : (M ou F).

```
10      INPUT S$
20      IF S$ = "F" OR S$ = "M" THEN 50
30      PRINT "ERREUR"
40      STOP
50      PRINT "OK"
60      END
```

– Soit à tester une condition négative telle que l'exclusion d'une condition. Ainsi, dans l'exemple précédent, aurait-on pu redemander l'entrée de la donnée dans le cas où elle est erronée.

```
10      INPUT S$
20      IF NOT S$ = "F" AND NOT S$ = "M" THEN 10
30      PRINT "OK"
40      END
```

Les instructions structurées dans certains BASIC

On a vu que l'instruction conditionnelle en BASIC est de type :

SI condition ALORS instructions.

Lorsque la condition est vérifiée on exécute la ou les instructions qui se trouvent après le ALORS sur la même ligne, tandis que si la condition n'est pas vérifiée, on passe à l'instruction en séquence.

De manière plus générale, une instruction conditionnelle structurée peut s'écrire :

SI condition ALORS instructions 1
SINON instructions 2

Certaines réalisations de BASIC permettent de définir des instructions de ce type. Dans ce cas, l'on s'écarte du standard BASIC. D'autre part, les groupes d'instructions 1 et 2 sont en

général encadrés par des structures indiquant explicitement le début et la fin de la série d'instructions, correspondant à chaque valeur de la condition.

On a alors une structure du type :

SI condition ALORS DEBUT instructions 1 FIN
SINON DEBUT instructions 2 FIN.

Ceci permet en particulier d'avoir des structures conditionnelles imbriquées du type :

```
SI condition 1 ALORS DEBUT
SI condition 2 ALORS DEBUT
        instructions. FIN
SINON DEBUT
        instructions FIN
SINON DEBUT
        instructions FIN.
```

Ainsi le BASIC développé par certains constructeurs permet des structures du type IF... THEN... ELSE.

La version la plus élaborée est disponible sur les systèmes BASIC développés par ZILOG sur les systèmes Z 80.

En effet, on y trouve des structures du type IF

```
IF condition THEN DO
Instructions
DOEND
ELSE DO
Instructions
DOEND
```

Exemple :

L'algorithme d'Euclide déjà vu ci-dessus s'écrirait alors :

```
 10    INPUT A, B
 20    IF A > B THEN DO
 30    LET C = A
 40    LET A = B
 50    LET B = C
 60    DOEND
 70    REM CALCUL DU PGCD
 80    LET Q = INT (A/B)
 90    LET R = A - B * Q
100    IF R< >O THEN DO
110    LET A = B
120    LET B = R
130    GO TO 80
140    DO END
150    ELSE DO PRINT B
```

```
160      DO END
170      END
```

Remarque. – Ce type de structure est surtout utile lorsque l'on a plusieurs conditions emboîtées les unes dans les autres.

Exercices et problèmes

– *Soit le programme BASIC suivant :*

```
10      INPUT X, Y
20      IF X < Y THEN PRINT X, "INFERIEUR", Y
30      IF X > Y THEN PRINT X, "SUPERIEUR", Y
40      IF X = Y THEN PRINT X, "EGAL", Y
50      GO TO 10
60      END
```

Ce programme est-il correct du point de vue syntaxique ? Que donnera-t-il comme résultat si on rentre X = 50 et Y = 100 ? Que donnera-t-il comme résultat si X = 20 et Y = 10 ? Dans quel cas fonctionne-t-il correctement ? Comment le modifier pour que les résultats attendus soient imprimés ?

– *Ecrire un programme qui rentre une série de valeurs non nulles et calculer la somme et la moyenne de ces nombres. La fin de la série de valeurs est représentée par la valeur 0.*

– *Ecrire un programme qui teste si une chaîne de caractères contient un autre chaîne (ou un mot).*

Solution des exercices

1° Ce programme est correct syntaxiquement.

- Pour X = 50 et Y = 100 on aura l'impression des trois messages successivement.
- Pour X = 20 Y = 10 on aura l'impression de deux messages.
- Le résultat est correct pour X = Y.
- Pour corriger ces erreurs, il suffit de rajouter les instructions :

```
25 GO TO 10
35 GO TO 10
```

2° Ce problème ne pose pas de difficultés particulières. Nous écrivons donc directement la solution. S contient la somme, N est le nombre d'éléments, M est la moyenne :

```
10      S = 0
20      N = 0
```

```
30    INPUT X
40    IF X = 0 THEN 80
50    S = S + X
60    N = N + 1
70    GO TO 30
80    M = S/N
90    PRINT "SOMME =" ; S ; "MOYENNE = " ; M
100   END
```

3° Il faut s'inspirer des exercices du paragraphe précédent recherchant une lettre dans une chaîne.

Ici, nous utiliserons un principe identique : on compare tous les caractères de la chaîne A$ au premier caractère de la chaine B$ (représenté par L$).

Lorsqu'il y a identité, on extrait alors de A$ une sous-chaîne M$ de longueur égale à B$ et s'il y a identité on incrémente un compteur. (On pourrait également imprimer un message.) On obtient :

```
1     N = 0
10    INPUT A$, B$
15    L$ = LEFT$ (B$, 1)
20    L1 = LEN (A$)
25    L2 = LEN (B$)
30    FOR I = 1 TO L1
40    C$ = MID$ (A$, I, 1)
50    IF C$ < > L$ THEN 60
52    M$ = MID$ (A$, I, L2)
55    IF M$ = B$ THEN N = N + 1
60    NEXT I
70    PRINT "LE MOT" ; B$ ; "APPARAIT" ; N ; "FOIS
      DANS" ; A$
80    END
```

4-5. L'instruction d'itération POUR (FOR)

Bien qu'avec les instructions d'affectation et de test on puisse programmer tous les algorithmes imaginables, l'on a vu que l'un des intérêts d'une machine est de pouvoir lui faire faire des travaux répétitifs et itératifs. Cela peut se faire de plusieurs manières :

- Si l'on connaît le nombre d'itérations, on procédera en initialisant un compteur d'itération, puis à chaque fois qu'une itération est terminée on l'incrémentera jusqu'à ce que l'on arrive au nombre prévu d'itérations.
- Dans d'autres cas, on ne connaît pas le nombre d'itérations, mais on connaît le critère d'arrêt. On peut dire que l'on itère

TANT QUE la condition est vérifiée ou que l'on REPETE les itérations JUSQU'A ce qu'une condition soit remplie (de telles structures sont par exemple disponibles dans un langage comme PASCAL).

En BASIC, seule la première structure a été prévue. C'est la structure de boucle de programme POUR... SUIVANT (FOR... NEXT). Pour ce qui est du deuxième type de structure, on peut cependant s'en tirer en prévoyant un nombre d'itérations très supérieur à celui escompté et en testant la condition constituant le critère d'arrêt à l'intérieur de la boucle. On peut alors sortir prématurément de celle-ci lorsque la condition est vérifiée.

Structure de l'instruction POUR en BASIC

Nous avons déjà vu des exemples dans les chapitres ou paragraphes précédents. Cependant, la forme générale est plus complexe.

POUR V = < expression arithmétique >$_1$ A < exp. arithm >$_2$
PAR PAS < exp. arith >$_3$
Suite d'instructions de l'itération
SUIVANT V.

En BASIC on a :

FOR V = < exp. arith > $_1$TO < exp. arith > $_2$STEP < exp. arith >$_3$

Instructions

NEXT V.

V est ce que l'on appelle la variable de contrôle :

– L'expression arithmétique 1 qui suit le mot clé FOR (POUR) représente la valeur initiale de la variable de contrôle de l'itération. En fait, l'instruction suivant le FOR peut être une instruction arithmétique d'affectation. Elle peut donc contenir des fonctions mathématiques.

– L'expression arithmétique 2 représente la valeur finale de la variable de contrôle.

– Le mot clé STEP (PAS) précise la valeur qu'il faut ajouter à chaque itération. C'est aussi une expression arithmétique. Le PAS peut donc être négatif ou fractionnaire.

Cependant ce paramètre peut être omis et dans ce cas le pas ou l'incrément est alors considéré comme égal à l'unité (1).

Instruction fin de boucle SUIVANT (NEXT)

L'introduction d'une instruction POUR (FOR) suppose que l'on indique là où doivent se terminer les instructions sur lesquelles doit porter l'itération. Ceci est fait grâce à l'instruction SUIVANT

(NEXT) qui est suivie de la variable de contrôle. Lorsqu'il n'y a pas d'ambiguïté (une seule boucle de programme) la variable de contrôle peut être omise après le NEXT.

Exemple : Nous reprenons ici l'exemple de la somme des N premiers nombres déjà présenté dans un paragraphe précédent. S est la somme et N est le nombre.

```
10      INPUT N
20      S = 0
30      FOR I = 1 TO N
40      S = S + I
50      NEXT I
60      PRINT S
70      END
```

Remarque. – Sur la plupart des BASIC, si l'on oublie l'initialisation S = 0, le programme fonctionnera quand même, car S est mis à zéro automatiquement.

Un autre exemple permet d'imprimer les résultats d'une fonction où le nombre de points N, l'incrément X et la valeur initiale sont des données rentrées au clavier. La valeur maximale est donc I + N × X.

On obtient :

```
10  INPUT I, N, X
20  FOR V = I TO I + N * X
    STEP X
30  PRINT V, LOG (V)
40  NEXT V
50  END
```

```
10  ENTRER I, N, X
20  POUR V = I A I + N*X
    PAR PAS X
30  IMPRIMER V, LOG (V)
40  SUIVANT V
50  FIN
```

Si l'on exécute ce programme avec les données I = 1, N = 5, X = 1 on obtient :

```
? 1, 5, 1
  1          0
  2          .693147181
  3          1.09861229
  4          1.38629436
  5          1.60943791
  6          1.79175947
```

On aurait pu faire exécuter le programme avec un pas fractionnaire (0.1) par exemple.

Structure itérative indéfinie (type 2)

On prendra l'exemple de la recherche du maximum d'une liste de nombres positifs rentrés au clavier : le dernier nombre est nul. Dans ce cas, on suppose que l'on itère jusqu'à ce que l'on rencontre un nombre nul.

```
10  M = 0                        10  M = 0
20  POUR I = 1 A 10000           20  FOR I = 1 TO 10000
30  ENTRER X                     30  INPUT X
40  SI X = 0 ALORS 70            40  IF X = 0 THEN 70
50  SI X > M ALORS M = X         50  IF X > M THEN M = X
60  SUIVANT                      60  NEXT I
70  IMPRIMER "MAXI-              70  PRINT "MAXIMUM = "
    MUM" ; M                         ; M
80  FIN                          80  END
```

Exécution de ce programme

```
RUN
?     2 ®
?    23 ®
?    98 ®
?     4 ®
?   456 ®
?     0 ®
□       MAXIMUM 456
```

Dans cet exemple il s'agit d'une structure itérative de deuxième type (TANT QUE ou REPETER JUSQU'A...). Ici, on a programmé une boucle de 1 à 10 000, mais dès que l'on a rentré un chiffre 0, on sort de la boucle et le maximum est alors imprimé.

Boucles POUR imbriquées

Jusqu'à présent nous avons vu simplement des exemples où un seul niveau de boucle était présent. Il est possible de définir à l'intérieur d'une première boucle, une deuxième boucle, éventuellement une troisième à l'intérieur de la deuxième, etc. On peut donc avoir des structures du type :

```
POUR I1 = ......
      POUR I2 = ......
            POUR IK = ......
            SUIVANT IK
            ......
      SUIVANT I2
SUIVANT I1
```

C'est ce que l'on appelle des boucles imbriquées. La seule règle à respecter est qu'il y ait emboîtement de toutes les boucles de niveau inférieur dans celles de niveau supérieur. Ainsi, il n'est pas possible d'avoir une structure du type :

```
POUR I1 = ......
      POUR I2......
SUIVANT I1
      SUIVANT I2
```

Règle 1. – Il ne peut y avoir de chevauchement de deux boucles POUR (FOR). La boucle la plus interne doit se terminer avant ou en même temps que la boucle externe. Dans le cas où les deux boucles se terminent au même endroit, on doit préciser dans l'instruction SUIVANT (NEXT) la variable de contrôle la plus interne d'abord et la plus externe ensuite.

Exemple :

```
10      A$ = "A"
20      B$ = "B"
30      FOR I = 1 TO 5
40      C$ = C$ + A$
50      FOR J = 1 TO 2
60      C$ = C$ + B$
70      PRINT C$
80      NEXT J, I
90      END
```

L'exécution de ce programme donne :

```
AB
ABB
ABBAB
ABBABB
ABBABBAB
ABBABBABB
ABBABBABBAB
ABBABBABBABB
ABBABBABBABBAB
ABBABBABBABBABB
```

Le même résultat aurait été obtenu si l'on avait utilisé :

```
75      NEXT J
80      NEXT I
```

Rentrée et sortie d'une boucle POUR (FOR)

On a vu sur l'exemple 3 précédent que la sortie d'une boucle avant sa terminaison est toujours possible. Par contre, on ne peut pas rentrer n'importe où dans une boucle. Bien que l'instruction GO TO permette théoriquement de se brancher n'importe où dans un programme, il est interdit de façon pratique de renvoyer à une instruction contenue dans une boucle FOR, à partir d'une instruction qui se trouve à l'extérieur de cette boucle.

Règle 2. – On ne peut rentrer dans une boucle POUR (FOR) que par l'instruction POUR (FOR) elle-même. On peut en sortir, par contre, à n'importe quel endroit.

Règles sémantiques concernant les paramètres d'une boucle POUR

Lorsque l'on a défini les paramètres d'une boucle on a vu que chacun pouvait être exprimé par une expression arithmétique. Or, si le paramètre incrément (le PAS) est *positif,* cela implique que la valeur de l'expression initiale de la variable de contrôle soit inférieure ou égale à la valeur terminale.

Inversement, si le PAS est négatif, il faut que la valeur initiale soit supérieure à la valeur terminale.

Règle 3. – Soient INIT et TERM les valeurs initiales et terminales :

- si PAS > 0 ALORS INIT ⩽ TERM
- si PAS < 0 ALORS INIT ⩾ TERM

En pratique, si l'on ne respecte pas cette règle, il faut savoir que dans la plupart des BASIC la boucle sera quand même exécutée *une* fois.

Exemple : Si l'on écrit le programme et qu'on l'exécute :

```
10      FOR I = 10 TO 1 STEP 1
20      PRINT I
30      NEXT I
40      END
```

On obtient

```
□ 10
```

Ce qui montre bien que l'on est passé une fois dans la boucle pour s'apercevoir que I ne pouvait pas varier. Ceci est une remarque importante, et beaucoup d'erreurs de programmation viennent du non-respect de cette règle.

Exercices

1. *Ecrire un programme qui calcule Factorielle N, c'est-à-dire le produit des N premiers nombres entiers.*
2. *Ecrire un programme qui calcule les nombres de Fibonacci représentés par la formule de récurrence.*

   $F_{n+2} = F_{n+1} + F_n$ *avec* $F_0 = 0, F_1 = 1, n \geqslant 0$

   Le début de la séquence est donc 0, 1, 1, 2, 3, 5, 8, 13...
3. *Ecrire un programme qui permette de calculer les intérêts composés mensuellement d'un capital C investi à un taux de I % annuel.*
4. *Ecrire un programme qui permette d'engendrer la séquence de lettres suivantes :*

   A
   AB
   ABA
   ABAB
   . . . . . .

Solutions

1. Il s'agit d'un programme similaire à celui de la somme.

```
10    F = 1
20    INPUT N
30    FOR I = 1 TO N
40    F = F * I
50    NEXT I
60    PRINT F
70    END
```

2.

```
10    F0 = 0
20    F1 = 1
30    INPUT N
40    FOR I = 1 TO N
50    F2 = F1 + F0
55    PRINT F2
60    F0 = F1
70    F1 = F2
80    NEXT I
90    END
```

Exécution

```
? 12 ®
  1
  2
  3
  5
  8
  13
  21
  34
  55
  89
  144
  233
```

3. Soit C le capital, I l'intérêt annuel et N le nombre de mois. On obtient :

```
10    INPUT C, I, N
20    PRINT "MOIS", "CAPITAL", "INTERET"
30    FOR J = 1 TO N
40    IN = C * I/100/12
50    PRINT J, C, IN
60    C = C + IN
70    NEXT J
80    END
```

Exécution

? 1000, 6, 6

MOIS	CAPITAL	INTERET
1	1000	5
2	1005	5.025
3	1010.025	5.050125
4	1015.07513	5.0753756
5	1020.1505	5.1007525
6	1025.25125	5.1262562

4.
```
 10  INPUT N
 20  C = 0
 30  FOR I = 1 TO N
 40  IF C = 0 THEN C = 1 : C$ = C$ + "A" : GO TO 70
 50  C = 0
 60  C$ = C$ + "B"
 70  PRINT C$
 80  NEXT I
100  END
```

La variable C prend ici alternativement les valeurs 0 et 1 pour indiquer le caractère à ajouter.

4-6. Les Instructions d'entrées-sorties :

Dans ce paragraphe, nous allons étudier les possibilités des instructions d'entrées-sorties : ENTRER (INPUT), IMPRIMER (PRINT), LIRE (READ) et OBTENIR (GET).

L'INSTRUCTION ENTRER (INPUT).

Cette instruction, dont on a vu l'utilisation dans de nombreux exemples, permet de rentrer des données numériques ou chaînes de caractères pendant l'exécution d'un programme. La syntaxe de cette instruction est très simple :

ENTRER (INPUT) liste de variables.

La liste des variables peut comprendre des noms de variables numériques ou non numériques, séparées par des virgules.

Exemple : ENTRER N, M, NO$, PR$
(INPUT)

Pendant l'exécution du programme, cette instruction provoquera l'émission d'un point d'interrogation : ?

C'est alors qu'il faut rentrer les données séparées par des virgules, la dernière étant suivie d'un Retour.

Exemple :

? 10, 15, DURAND, MARIE ®

On a donc :

N = 10 M = 15 NO$ = "DURAND" PR$ = "MARIE".

Remarque. – Lorsqu'une donnée alphanumérique est rentrée par un ordre INPUT, il n'est pas nécessaire d'utiliser de guillemets pour indiquer le contenu de la chaîne de caractères. Mais si l'on veut faire entrer des blancs à gauche, il faudra utiliser des guillemets. Cela suppose donc lorsque l'on rentre une suite de données, que l'on sache ce que le programme attend et dans quel ordre. Ainsi, de façon réaliste, si le programme est destiné à des utilisateurs ne connaissant pas le programme, il faut indiquer clairement ce que l'on attend avant de demander une entrée de données (voir ci-dessous l'utilisation de libellés).

Les erreurs qui peuvent se produire à l'exécution d'un ordre ENTRER (INPUT)

– Si l'on ne rentre pas assez de données et que l'on frappe un retour ®, la réponse sera un double point d'interrogation indiquant qu'il manque des données : ??

– Si par contre, le nombre de données rentrées est supérieur à celui attendu, il n'y a pas toujours de messages, et les données en trop sont ignorées. Lorsqu'il y a un message, il sera sur les systèmes anglophones du type : ? EXTRA IGNORED (Donnée superflue ignorée).

– Si l'on rentre des données dont le type ne correspond pas à celui attendu (chaîne de caractères au lieu de chiffres par exemple), dans ce cas il y a en général un message d'erreur et l'entrée est demandée à nouveau. Sur les systèmes anglophones les messages seront du type "INPUT ERROR TYPE", "ERROR TYPE MISMATCH" ? REDO FROM START (Retour au début de l'entrée).

Impression d'un libellé avant l'entrée d'une donnée :

A l'aide de l'ordre ENTRER (INPUT), il est possible de prévoir un libellé précédant le point d'interrogation. Pour cela, il suffit de faire suivre l'ordre INPUT d'une constante chaîne de caractères suivie d'un ; et de la liste de variables.

On a donc une structure du type :

```
10      ENTRER "LIBELLE" ; liste de variables.
        (INPUT)
```

Ceci provoquera à l'exécution l'impression : LIBELLE ? ...

Exemple :

Soit à rentrer trois données numériques correspondant à une date de naissance. On pourra écrire :

10 ENTRER "DATE DE NAISSANCE" J, M, A
(INPUT)

□ DATE DE NAISSANCE ? 15, 2, 68 ®

Ceci permet donc de préciser pour chaque ordre ENTRER (INPUT) la nature de la donnée attendue par le programme.

Règles concernant l'ordre ENTRER :

On peut résumer les règles concernant l'ordre ENTRER (INPUT) par les trois règles suivantes :

Règle 1. – Les données rentrées doivent correspondre à l'ordre de la liste de variables et au type correspondant à chaque variable. Lorsqu'il y a plusieurs données, elles doivent être séparées par des virgules (,).

Règle 2. – Si l'on désire une impression de libellé, il doit suivre le mot clé INPUT (ENTRER) et se terminer par un ;.

Règle 3. – Si l'on désire rentrer des chaînes de caractères, les blancs à gauche ou à droite sont supprimés. Si l'on désire les prendre en compte, il faut utiliser des guillemets. Les guillemets sont également obligatoires si la chaîne de caractères contient une virgule.

L'ORDRE OBTENIR (GET)

Cette instruction n'est pas disponible sur tous les BASIC. Cet ordre permet de lire un seul caractère à la fois.

La syntaxe de l'ordre GET est similaire à celle de l'ordre ENTRER (INPUT)

GET liste de variables.

Cependant, en pratique, comme un seul caractère est lu à la fois, il est nécessaire d'avoir une seule variable. De plus comme cette instruction est faite pour tester l'entrée de n'importe quel caractère au clavier, il est préférable d'utiliser une variable chaîne de caractères. En effet, l'utilisation d'une variable numérique provoquera une erreur d'exécution si l'on rentre une autre valeur. D'autre part, dans ce cas, si aucun caractère n'est rentré, la valeur correspondante sera nulle, ce qui ne permet donc pas de différencier la rentrée du caractère 0 de l'absence de caractère. Il est important de bien souligner le caractère *"Temps réel"* de l'instruction GET.

En effet, contrairement à l'ordre INPUT, cette instruction *n'attend pas* que l'on ait rentré un caractère pour continuer à exécuter le programme. Ainsi, une instruction GET ne se conçoit guère sans boucle de scrutation de l'arrivée ou de l'absence d'un caractère.

Remarque : Sur certaines réalisations, l'instruction GET *attend* l'entrée d'un caractère.

Exemple :

Dans le premier cas, si l'on désire entrer un caractère, il faut donc boucler sur l'instruction GET tant que l'on n'a pas obtenu un caractère.

```
10      GET X$
20      IF X$ = " " THEN 10
30      PRINT X$
40      END
```

L'instruction GET est en particulier très utile pour les entrées interactives de type temps réel, notamment dans les jeux où l'écran ne doit pas être détruit par une entrée clavier qui est cependant nécessaire pour indiquer le choix du joueur.

L'INSTRUCTION LIRE (READ)

Jusqu'à présent, nous n'avons jamais parlé de cette instruction. La raison en est que cette instruction introduit une confusion dans la notion de données. D'autre part, cette instruction, qui fait partie du langage, est en fait un anachronisme résultant du fait que lors de l'introduction du BASIC peu de machines disposaient de systèmes interactifs ou de temps partagé. Ainsi, pouvait-on développer des programmes et les faire exécuter de manière non interactive en introduisant des instructions READ (LIRE) et des instructions DATA (DONNEE) associées.

La syntaxe de l'instruction LIRE (READ) est également très simple.

LIRE liste de variables
(READ)

Là aussi, la liste de variables peut être quelconque (numérique et/ou chaînes de caractères), chacune des variables étant séparées par une virgule.

Exemple : LIRE A, B$, C, D
(READ)

Les déclarations de données (DATA) associées à l'ordre READ (LIRE)

Une instruction LIRE suppose toujours l'association d'une instruction de déclaration de données.

En effet, dans le cas d'une instruction READ (LIRE), les données sont incorporées dans le programme sous forme de déclaration

DATA (DONNEE), donnant les valeurs des données associées aux variables de l'ordre READ. La syntaxe de la déclaration est

DATA suite de valeurs
(DONNEE)

Exemple :

```
10      LIRE, A, B,  C, D
        (READ)
20      DATA 7, 15.5, 1.3 E - 2, - 54.36
```

Les données sont également séparées par des virgules et les valeurs sont associées aux variables correspondantes définies dans l'instruction LIRE.

Dans l'exemple ci-dessus on aura A = 7, B = 15,5, C = 1.3 E - 2, D = - 54.36.

L'instruction DATA peut ne pas suivre l'ordre LIRE et être n'importe où dans le programme. Il peut y avoir plusieurs instructions DATA pour une même instruction LIRE (READ), mais, dans ce cas, la lecture commencera par les données de la première instruction DATA. Les autres instructions DATA seront considérées comme la suite de la première dans l'ordre où elles apparaissent dans le programme. Lorsqu'une donnée a été lue, elle ne peut être relue (sauf si l'on utilise l'instruction RELIRE : cf. ci-dessous). Chaque lecture fait donc progresser la lecture dans la liste des données disponibles dans un programme.

Dans le cas de données chaînes de caractères, comme pour l'introduction INPUT, il n'est pas nécessaire de mettre des guillemets, sauf si l'on veut prendre en compte les blancs au début ou à la fin de la chaîne ou si l'on veut prendre en compte une virgule.

Exemple : (LIRE)

```
READ A$, B$, C$
DATA TEXTE, "ƀMOTƀ", "MOT 1, MOT 2"
(DONNEE)
```

Dans ce cas, on a :

```
A$ = TEXTE
B$ = ƀ MOT ƀ
C$ = MOT 1, MOT 2
```

Là aussi, si l'on demande de lire plus de variables qu'il n'y a de données, il y aura un message d'erreur du type "? ERREUR MANQUE DES DONNEES" ("?OUT OF DATA ERROR" pour les systèmes anglophones).

En fait, lors de l'exécution, l'interpréteur met à jour un "pointeur" qui indique l'adresse mémoire de la prochaine donnée à

lire. Lorsque la liste est épuisée ; le pointeur adresse un symbole spécial appelé *terminateur,* qui indique qu'il n'y a plus de donnée.

L'instruction RELIRE (RESTORE)

Cette instruction permet précisément de relire une liste de données en réinitialisant le pointeur au début de la liste de données. Il suffit pour cela d'insérer une instruction : RELIRE (RESTORE).

Cette instruction permet donc de préparer la relecture des *mêmes* données, mais le traitement peut être différent. Par contre, ce n'est pas cette instruction qui permet de lire ces données. Il faut pour cela utiliser une instruction LIRE (READ), comme dans le cas de la première lecture.

Exemple :

```
  1  DONNEE 8, 4, 7, 3, 5, 12, 9, 14, -3, 17   DATA
 10  S = 0                                      S = 0
 20  POUR I = 1 A 10                            FOR I = 1 TO 10
 30  LIRE X                                     READ X
 40  S = S + X                                  S = S + X
 50  SUIVANT                                    NEXT
 60  IMPRIMER S                                 PRINT S
 70  P = 1                                      P = 1
 80  RELIRE                                     RESTORE
 90  POUR I = 1 A 10                            FOR I = 1 TO 10
100  LIRE X                                     READ X
110  P = P * X                                  P = P * X
120  SUIVANT                                    NEXT
130  IMPRIMER P                                 PRINT P
140  FIN                                        END
```

La première partie du programme effectue la somme des dix nombres donnés et la deuxième effectue le produit de ces mêmes nombres. Bien sûr, cet exemple n'est utilisé que pour noter l'utilisation de l'instruction RELIRE. On aurait pu l'éviter si l'on avait regroupé les deux boucles POUR dans une seule effectuant la somme et le produit simultanément.

Remarques et précautions à prendre pour l'utilisation de l'instruction LIRE

On a vu que l'instruction LIRE (READ) n'apportait rien de plus que l'instruction ENTRER (INPUT), d'autre part elle suppose que les données soient connues au moment de l'écriture du programme. Ainsi ce type d'instruction d'entrée est absolument à proscrire si

l'on désire faire utiliser un programme par des utilisateurs ne connaissant pas le langage. En effet, dans ce cas, la porte est ouverte à des modifications non désirées du programme (erreur de frappe, fausse manipulation, etc.).

L'ordre LIRE n'est vraiment utile que lorsqu'il s'agit de lire des données qui sont stables ou relativement stables (c'est-à-dire modifiées peu souvent par rapport au nombre de fois où le programme est utilisé). Il peut s'agir de constantes ou de paramètres fixes pendant un certain temps : ainsi les prix unitaires de produits qui ne varient pas tous les jours..., etc. Dans ce cas, et dans ce cas seulement, on aura intérêt à utiliser l'instruction LIRE.

D'autre part, il s'agit d'une mauvaise pratique que d'utiliser une instruction LIRE pour initialiser une variable qui sera ultérieurement modifiée dans le cours du programme. En effet, cela risque d'entraîner des erreurs lorsque l'on veut revenir au début de l'algorithme correspondant.

L'INSTRUCTION IMPRIMER (PRINT)

Il s'agit en fait de l'instruction de SORTIE de résultats qui le plus souvent sont imprimés, d'où le nom de l'instruction.

Cependant, dans le cas d'un système micro-ordinateur, cette « impression » se fait le plus souvent sur un écran de visualisation. Ce n'est donc que pour des raisons historiques que l'on maintient le mot clé IMPRIMER (PRINT). Il vaudrait mieux utiliser le verbe SORTIR (OUTPUT).

La syntaxe de l'instruction est également très simple :

IMPRIMER liste d'expressions
(PRINT)

Dans la liste, les expressions peuvent contenir soit des variables, des constantes numériques et / ou des chaînes de caractères, soit des fonctions mathématiques ou chaînes de caractères. La séparation des différents éléments de la liste se fait soit par des virgules (,), soit par des points virgules (;), mais cela n'a pas le même effet.

– Si le séparateur est une *virgule,* la donnée sera imprimée en commençant à des endroits fixes de la ligne, par exemple tous les dix caractères.

– Si le séparateur est un *point virgule,* les données seront imprimées à la suite les unes des autres. Ceci implique que l'on introduise des caractères blancs pour séparer les résultats.

Remarque : Sur certaines machines, on peut utiliser le caractère ? à la place du mot clé PRINT

Exemple :

? 5 + 6® est équivalent à PRINT 5 + 6® □ 11

Règles concernant l'instruction IMPRIMER (PRINT)

– L'instruction PRINT provoque l'impression d'au moins une ligne.

– Une instruction PRINT sans liste consécutive produit une ligne blanche.

– Dans la plupart des BASIC, une donnée numérique de moins de huit chiffres significatifs est imprimée sous la forme décimale (± XXX.XXX).

Si la donnée contient plus de huit chiffres ou est inférieure à 10^{-8}, elle est imprimée sous la forme flottante avec mantisse et exposant : ± XXXX.XXXX E ± XX.

– Les chaînes de caractères doivent toujours être entre guillemets.

– Si la liste à imprimer se termine par un point virgule (;), le caractère suivant (autre ordre PRINT ou ? envoyés lors d'un ordre INPUT) est imprimé sur la même ligne.

Exemple :

– 10	IMPRIMER "NOM" ;	PRINT "NOM" ;
20	ENTRER N$	INPUT N$

provoque l'impression de
□ NOM ?

Le ? est en fait imprimé pour l'ordre ENTRER. Ces deux instructions sont donc équivalentes à :
10 ENTRER "NOM" ; N$

La fonction de TABULATION (TAB)

Il s'agit d'une fonction qui est disponible sur la plupart des BASIC et qui permet de réaliser des mises en pages et des éditions. Elle ne peut être utilisée que dans une instruction d'impression (PRINT).

La syntaxe de cette fonction est très simple.

TAB (N)

Le paramètre N est une variable ou constante entière qui spécifie la position ou l'on doit commencer une impression.

Exemples :

– 10 PRINT TAB (4) ; "MARGE DE 4"

Cette instruction donnera une marge de quatre blancs, puis imprimera le message entre guillemets.

□ MARGE DE 4

```
- 10    FOR I = 1 TO 10
- 20    PRINT TAB (I) ; I ; "BLANCS"
- 30    NEXT I
```

Exécution :

```
 1 BLANCS
  2 BLANCS
   3 BLANCS
    4 BLANCS
     5 BLANCS
      6 BLANCS
       7 BLANCS
        8 BLANCS
         9 BLANCS
          10 BLANCS
```

Si la fonction TAB est utilisée plusieurs fois dans la même instruction, le paramètre N correspond à la position N depuis le début de la ligne et non pas depuis le dernier caractère imprimé.

Exemples :

– PRINT TAB (4) ; "*" ; TAB (5) ; "*"

donnera comme résultat :

□ **

– PRINT TAB (5) ; "*" ; TAB (5) ; " "

donnera comme résultat :

□ *

Ceci montre qu'il n'y a pas superposition de deux impressions, même si elles se font en utilisant la même tabulation. Ceci est vrai également si la position de la tabulation tombe sur un caractère déjà imprimé.

– PRINT TAB (5) ; "TOTO" ; TAB (7) ; "TITI"

donnera

□ TOTOTITI

De même, sur la plupart des BASIC, si l'on utilise deux tabulations dans un ordre différent de celui attendu, tout se passe comme si la deuxième tabulation était ignorée.

– PRINT TAB (10) ; "TOTO" ; TAB (2) ; "TITI"

donnera le même résultat que ci-dessus

□ TOTOTITI

La fonction TAB peut également être utilisée avec une expression arithmétique comme paramètre. Dans ce cas c'est la partie entière de la valeur absolue de l'expression qui est utilisée pour définir la position de la tabulation.

Exemples :

```
 – 10        X = 3.14
   20        PRINT TAB (X) ; X
   30        END
```

donne une tabulation de 3 (partie entière de X)

```
   □         3.14
```

```
 – 10        X = 3
   20        PRINT TAB (X ↑2) ; X
```

donne une tabulation de 9

```
   □                   3
```

```
 – 10        X = 2 : Y = 9
   20        PRINT TAB (X ↑2 + SQR (Y)) ; X ; Y
```

donne une tabulation de 7

```
   □                2     9
```

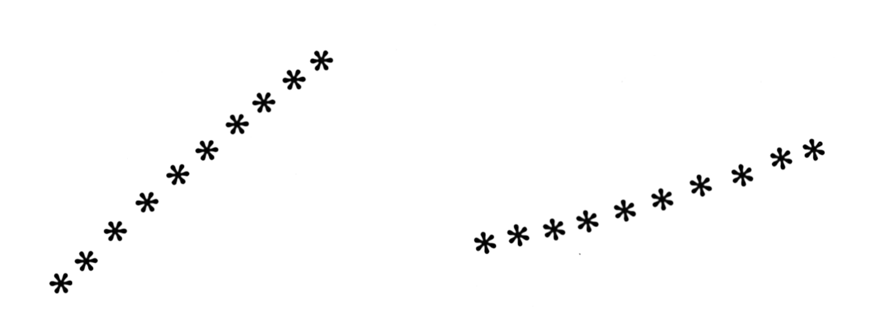
Ceci permet de faire des tracés *approximatifs* points par points de fonctions en ne considérant que les valeurs entières positives.

Exemple : Tracé d'une droite :

```
10       INPUT A, B
20       FOR I = 10 TO 1 STEP – 1
30       PRINT TAB (A * I + B) ; "*"
40       NEXT I
```

Exécution :

? 1, 2 ®   ? 2, 3 ®

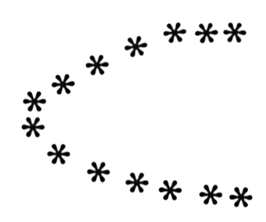


pente 1   pente 2

Autres exemples :

```
– 1     INPUT A
  10    FOR I = 10 TO 1 STEP – 1
  20    PRINT TAB (I↑2/A) ; "*"
  30    NEXT I
  40    FOR I = 1 TO 10
  50    PRINT TAB (I↑2/A) ; "*"
  60    NEXT I
```

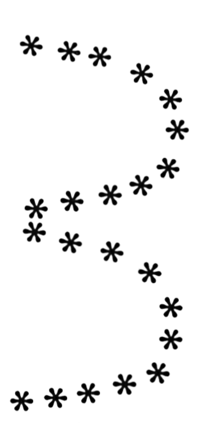
permet d'obtenir une allure de courbe parabolique :



```
– 10    FOR I = 1 TO 10 STEP 0.1
  20    PRINT TAB (ABS (SIN(I))*40) ; "*"
  30    NEXT I
```

permet d'obtenir une allure de la fonction

| SIN (I) |



Exercices

1. *Ecrire un programme permettant de rentrer des adresses postales sous la forme :*
   NOM ? PRENOM ?
   N° ET RUE ?
   CODE POSTAL ?
   VILLE ?
   et imprimer une adresse sous la forme :
      DUPOND JEAN
      ligne blanche
      18 RUE DES ACACIAS
      ligne blanche
      75015 PARIS
2. *Lire une liste de libellés de produits (maximum de 5 produits) et des prix unitaires associés.*
   Entrer un NOM de produit, la quantité et imprimer le prix total.
3. *Ecrire un programme similaire au précédent en utilisant des instructions GET pour rentrer un code de produits et le nombre d'unités. On pourra par exemple visualiser la liste des produits :*

   ```
   1  PRODUIT  1        2  PRODUIT  2
   3  PRODUIT  3        4  PRODUIT  4
   ```

   Après rentrée du code, on affiche le code sélection, le nombre d'unités et le prix sous la forme :

   ```
   1    *    5    PRIX TOTAL = XX.XX
   ```

CORRIGES

1.
```
10   INPUT "NOM PRENOM" ; N$
20   INPUT "N° ET RUE" ; AD$
30   INPUT "CODE POSTAL" ; C
40   INPUT "VILLE" ; V$
50   PRINT N$
60   PRINT
70   PRINT AD$
80   PRINT
90   PRINT C ; "   " ; V$
100  END
```

2.
```
10   READ P1$, P1, P2$, P2, P3$, P3, P4$, P4, P5$, P5
20   DATA CROISSANT, 2.0, ECLAIR, 4, BABA, 2.5,
     PARIS-BREST, 2.3, RELIGIEUSE, 3
```

```
30      INPUT "GATEAU" ; G$
40      INPUT "NOMBRE" ; N
50      IF G$ = P1$ THEN P = P1 * N
60      IF G$ = P2$ THEN P = P2 * N
70      IF G$ = P2$ THEN P = P3 * N
80      IF G$ = P4$ THEN P = P4 * N
90      IF G$ = P5$ THEN P = P5 * N
100     PRINT
110     PRINT "PRIX TOTAL =" ; P
120     END
```

Remarque. – Ce corrigé n'est pas la façon la plus concise de programmer le problème, et elle tient compte uniquement de ce qui a été vu jusqu'à ce point. Nous conseillons au lecteur de reprendre ce problème après avoir vu le chapitre sur les listes et tableaux.

Le corrigé de l'exercice n° 3 reprend l'exemple du 2, mais en supposant que l'on rentre des caractères purement numériques identifiant le produit et le nombre d'unités.

Nous supposons que l'on dispose d'une commande permettant d'effacer tout l'écran. (HOME ou caractère spécial ♡).

```
1       REM LIRE LES PRIX UNITAIRES
10      READ P1, P2, P3, P4, P5
20      DATA 1.4, 2.5, 2.8, 3, 2.5
25      REM EFFACER L'ECRAN
30      PRINT "♡ " (HOME)
35      REM VISUALISER LES PRODUITS
40      PRINT "1 CROISSANT", "2 BABA"
45      PRINT
50      PRINT "3 ECLAIR", "4 PARIS BREST"
55      PRINT
60      PRINT "5 RELIGIEUSE"
65      PRINT
70      REM ENTRER LE CODE
80      GET G$
90      IF G$ = " " ; THEN 80
100     PRINT " " ; G$
110     REM ENTREE DU NOMBRE
120     GET N
130     IF N = 0 THEN 120
135     REM CALCUL DU PRIX
140     IF G$ = "1" THEN P = P1 * N
150     IF G$ = "2" THEN P = P2 * N
160     IF G$ = "3" THEN P = P3 * N
170     IF G$ = "4" THEN P = P4 * N
180     IF G$ = "5" THEN P = P5 * N
190     PRINT "PRIX TOTAL =" ; P
```

```
195    REM ATTENDRE LE RETOUR POUR PASSER AU
       SUIVANT
200    GET R$
210    IF R$ = " " THEN 200
220    IF ASC (R$)< > 13 THEN 200
230    RESTORE
250    GO TO 10
```

L'exécution de ce programme donne sur l'écran :

```
1    CROISSANT          2    BABA
3    ECLAIR             4    PARIS BREST
5    RELIGIEUSE
```

L'entrée de 2 suivi de 3 donne :

```
2    *    3    PRIX TOTAL = 7.5
```

Soit le produit 2 (BABA) demandé trois fois, ce qui donne bien un prix total de 7.5.

* Ce programme a été testé sur une machine PET de Commodore.

Exercices non corrigés

1. *Il y a eu beaucoup d'essais pour construire des formules qui génèrent des nombres premiers.*
   Ainsi, il existe beaucoup de nombres premiers de la forme :
   $$n^n + 1 \; (1^1 + 1 = 2 \qquad 2^2 + 1 = 5).$$
   Ecrire un programme BASIC qui détermine si $n^n + 1$ *est un nombre premier pour* n *variant de 1 à 7.*
2. *La formule* $x_n = n^2 + n + 41$ *permet de générer des nombres premiers pour* n = *1, 2, 3, 4...*
   Ecrire un programme qui utilise ce générateur de nombres premiers de n = *1 à 43 et déterminer par programme si le nombre obtenu* x_n *est premier.*
3. Ecrire un programme qui permet de calculer le nombre 2^{200} *(nombre très grand) et de l'imprimer.*
4. *Soit la suite G*i *définie par :*
   $$G_i = \frac{\sqrt{5}}{5}\left[\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i\right]$$
   Ecrire un programme BASIC calculant
   $$M_i = \text{INT}(G_i + 0.5)$$
   i *variant de 1 à 20 ; INT désigne la partie entière.*
5. *Vérifier que la suite M donne les nombres de Fibonacci :*
   $$(F_i = F_{i-1} + F_{i-2}) \qquad F1 = F2 = 1$$

CONCLUSION DU CHAPITRE III

Dans ce chapitre, nous avons abordé en détail toutes les instructions de base du langage : les instructions arithmétiques, de traitement de chaînes de caractères, de test, d'itération et d'entrées-sorties. Nous conseillons au lecteur débutant de bien assimiler ce chapitre avant de passer à la suite.

Dans la suite, nous allons aborder des techniques plus évoluées, notamment le traitement des listes, des tableaux, les instructions d'entrées-sorties à partir de fichiers, les instructions de manipulation d'écran...

Nous donnerons également des exemples plus élaborés, et il est important d'avoir bien assimilé les exercices du chapitre III.

CHAPITRE IV

LES LISTES - LES TABLEAUX
LES FONCTIONS
LES SOUS-PROGRAMMES
LES AIGUILLAGES

1. TRAITEMENT DES LISTES ET TABLEAUX EN BASIQUE-BASIC

Dans ce chapitre nous allons prolonger l'étude des éléments de base du langage, notamment en ce qui concerne le traitement des listes, des tableaux. Nous mentionnerons également les possibilités de traitement direct de matrices sur certains BASIC.

Dans la deuxième partie du chapitre nous étudierons les fonctions et les sous-programmes. Enfin, nous terminerons par l'étude des instructions d'aiguillage.

1-1. Les notions de liste et de tableaux

Jusqu'à présent les éléments de base du langage ont été des variables simples qui ne contiennent qu'une seule valeur à un instant donné.

Or, il est souvent nécessaire de pouvoir traiter des ensembles de données structurées : ainsi une suite de nombres a_i ou une suite de mots d'un dictionnaire m_i..., etc.

Lorsque l'on traite des problèmes mathématiques, il est également très courant d'avoir besoin de structure rectangulaire ou carrée avec des lignes et des colonnes et des éléments du type a_{ij}.

En mathématiques des variables du type a_i, a_{ij} sont ce que l'on appelle des éléments indicés ou génériques.

NOTION DE VARIABLE INDICEE

En programmation, il n'est pas possible de définir des indices par un artifice typographique. Il est par contre possible de définir des indices associés à des noms de variables. Dans ce cas on parlera de variables indicées. Ainsi a_i est représenté par A(I), où A est un identificateur et I est également un identificateur numérique. A(I) est une variable indicée : A(I) représente l'élément générique d'une *liste* de valeurs numériques : A(1), A(2), A(3)... A(K)... A(N). Ici N est l'indice associé au dernier élément de la liste. En effet, en programmation, il n'existe pas de liste infinie, d'où la nécessité de dimensionner le nombre maximum d'éléments d'une liste.

De même, si l'on veut représenter un élément à double indice, tel que b_{ij}, on définit une variable indicée B(I,J) où B est un identificateur de variable et I, J sont des variables numériques *entières*. Dans ce cas B(I,J) représente l'élément générique d'une table ou tableau : B(1,1), B(1,2), B(1,3)... B(1,N), B(2,1)... B(2,N)... B(M,1)... B(M,N).

Dans cet exemple, N est l'indice représentant la dernière colonne du tableau et M est l'indice représentant la dernière ligne du tableau. Ainsi, les éléments du tableau peuvent être représentés par :

B(1,1) B(1,2) . . . . B(1,N)
B(2,1) B(2,2) . . . . B(2,N)
. . . .
B(M,1) B(M,2) . . . . B(M,N)

Dans le cas où N = M, on a un tableau carré.

La notion de tableau permet bien sûr de représenter des matrices, mais il ne faut pas confondre les deux notions, car un tableau peut contenir des éléments qui ne sont pas des coefficients matriciels. D'autre part, il est possible de définir des listes et des tableaux de chaînes de caractères. Pour cela, il suffit d'utiliser un identificateur qui se termine par le caractère $.

Exemples :

– L$(K) est une variable indicée représentant l'élément générique d'une liste de chaîne de caractères.

L$(1) est le premier élément, L$(2) le deuxième...

– M$(K,J) est une variable indicée représentant l'élément générique d'un tableau de chaînes de caractères.

Remarque. – Dans certains cas une liste est appelée un tableau à une dimension, car il y a un seul indice qui varie.

La déclaration DIMENSION

Cette déclaration est pratiquement la seule instruction de déclaration du langage BASIC (il y a aussi la déclaration DATA que l'on a déjà vu). Elle permet de définir le nombre d'éléments maximum d'une liste ou d'un tableau. En effet, l'utilisation de listes ou de tables suppose que l'interpréteur réserve de la place en mémoire pour stocker les différents éléments des listes ou tableaux à traiter. Ceci est effectué par une déclaration qui doit *précéder* l'utilisation d'une liste ou d'un tableau : c'est la déclaration DIM.

La syntaxe en est très simple, puisqu'il suffit de définir les noms des listes ou tableaux avec entre parenthèses la ou les dimensions maximum des différents éléments.

Exemple :

– DIM A(10), A$(20)

La première liste contient dix éléments numériques, la deuxième 20 éléments chaîne de caractères.

– DIM T(10,10), TA$(5,10)

Le premier tableau contient 100 éléments numériques organisés en 10 lignes et 10 colonnes. Le deuxième tableau est un tableau de chaîne de caractères organisé en 10 lignes et 5 colonnes.

Remarques. – Sur la plupart des BASIC, l'instruction de déclaration DIM n'est obligatoire que lorsque le nombre d'éléments, d'une liste ou d'un tableau est supérieur à 10. Ainsi, pour tester des programmes, peut-on travailler sans définir de déclaration DIM.

Dimensions variables

Il est possible de définir la dimension par une variable numérique considérée comme une donnée qui sera lue au moment de l'exécution.

Exemple :

```
10    INPUT    N
20    DIM      B(N)
```

Ces instructions permettent de définir une liste B de dimension variable, N étant une donnée rentrée au moment de l'exécution et précisant la taille de la liste à ce moment-là.

Champ de variation des indices

Il existe cependant une contrainte concernant les indices. Ils doivent être *entiers* et *positifs ou nuls.*

D'autre part, évidemment, ils doivent avoir une valeur maximale qui ne dépasse ni la dimension maximale définie ni la taille mémoire disponible !

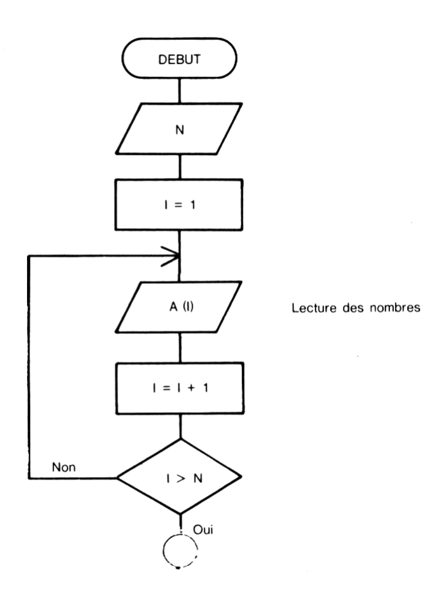
1-2. Le traitement des listes et des tableaux

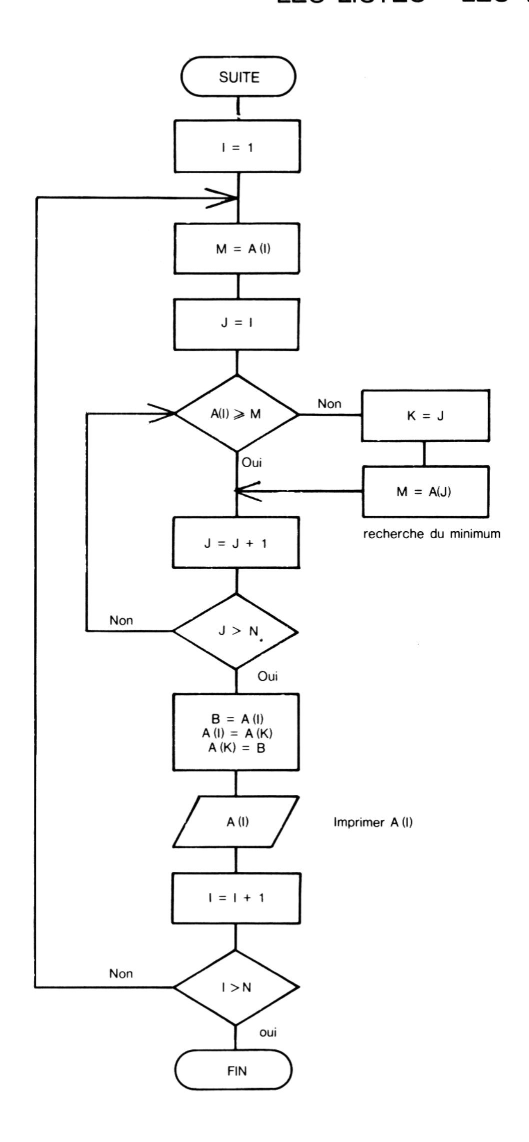
On a vu que grâce à l'ordre DIMENSION (facultatif pour les petites listes ou tableaux) il est possible de définir des structures de listes ou de tables. Néanmoins, lorsque l'on veut traiter de telles entités il faudra les traiter élément par élément. Ceci suppose donc l'utilisation des instructions itératives POUR... Dans ce paragraphe nous allons étudier sur des exemples l'utilisation de telles structures.

UN PROGRAMME DE TRI

Soit à résoudre le problème du classement d'une suite de nombres par ordre croissant. Il existe plusieurs algorithmes de tri : l'un des plus simples, mais l'un des moins efficaces en temps machine, est celui qui consiste à rechercher le minimum et à le placer comme premier élément de la liste, puis à rechercher le minimum suivant et à le placer en deuxième..., etc.

On suppose que l'on rentre le nombre d'éléments de la liste, puis les éléments eux-mêmes. L'organigramme de ce programme est alors :





Organigramme programme de tri

Le programme correspondant est alors :

```
  1  REM ENTREE DES NOMBRES
 10  INPUT N
 20  DIM A(N)
 30  FOR I = 1 TO N
 40  INPUT A(I)
 50  NEXT I
 55  REM PROGRAMME DE TRI
 60  FOR I = 1 TO N
 70  M =  A(I)
 80  FOR J =  I TO N
 90  IF A(J)< M THEN K = J : M = A(J)
100  NEXT J
110  B = A(I) : A(I) = A(K) : A(K) = B
120  PRINT A(I)
130  NEXT I
140  END
```

Dans cet exemple, nous avons utilisé la possibilité d'instructions multiples sur la même ligne ; ce qui rend le programme plus lisible et plus compact. En basique, on aurait le programme ci-après, ce qui constitue un programme très lisible ou l'algorithme utilisé est évident :

```
 10  ENTRER N
 20  DIM A(N)
 30  POUR I = 1 A N
 40  ENTRER A(I)
 50  SUIVANT
 60  POUR I = 1 A N
 70  M = A (I)
 80          POUR J = I A N
 90          SI A(J)< M ALORS K = J : M = A(J)
100          SUIVANT
110  B = A(I) : A(I) = A(K) : A(K) = B
120  IMPRIMER A(I) ;
130  SUIVANT
140  FIN
```

Un autre algorithme de tri

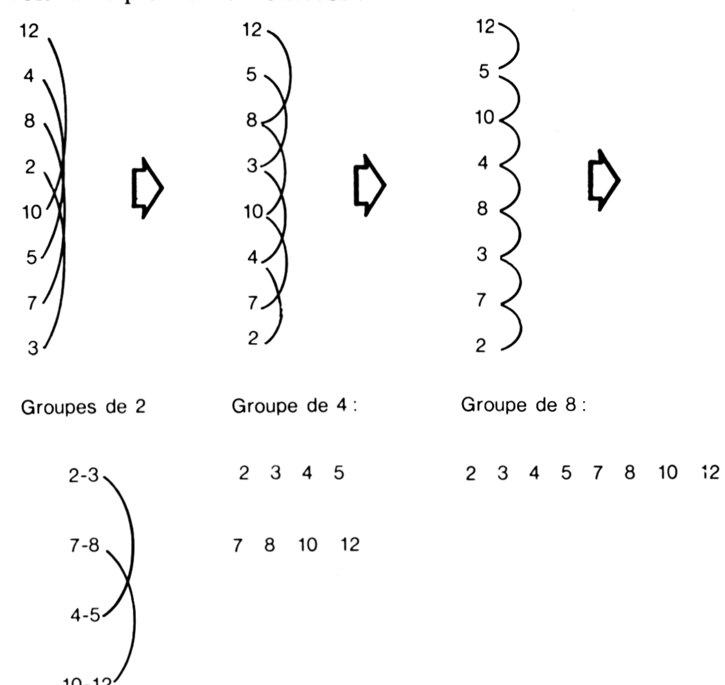
L'exemple d'algorithme utilisé précédemment est très inefficace, car il faut N^2 comparaisons pour trier une liste de nombres. Il existe un autre algorithme du même type, que l'on appelle le « tri à bulles », qui est proposé en exercice.

Ici, nous nous proposons d'étudier un autre algorithme de tri que l'on nomme l'algorithme de Shell.

Le principe en est le suivant : on compare d'abord les éléments deux à deux : B(I) avec B(I + N/2) et on les ordonne. Ensuite on fusionne les groupes de deux éléments en groupes de quatre éléments ordonnés, puis les groupes de quatre en groupes de huit..., jusqu'à ce que l'on arrive au groupe classé complet.

Exemple :

Soit la séquence de nombres :



Le programme correspondant est donné ci-dessous :

```
 10   ENTRER N               INPUT N
 20   DIM B (N)              DIM B (N)
 30   POUR I = 1 A N         FOR I = 1 TO N
 40   ENTRER B (I)           INPUT B (I)
 50   SUIVANT                NEXT
 60   L = N                  L = N
 70   L = ENT (L/2)          L = INT (L/2)
 80   SI L = 0 ALORS 190     IF L = 0 THEN 190
 90   POUR J = 1 A N-L       FOR J = 1 TO N-L
100   I = J                  I = J
110   SI B (I)> = B (I + L)  IF B (I)> = B (I + L)
      ALORS 170              THEN 170
```

```
120   C = B (I)                C = B (I)
130   B (I) = B (I + L)        B (I) = B (I + L)
140   B (I + L) = C            B (I + L) = C
150   I = I - L                I = I - L
160   SI I> = 1 ALORS 110      IF I> = 1 THEN 110
170   SUIVANT J                NEXT J
180   ALLER A 70               GO TO 70
190   POUR I = 1 A N           FOR I = 1 TO N
200   IMPRIMER B (I);          PRINT B (I);
210   SUIVANT                  NEXT
220   FIN                      END
```

Remarque. – La boucle J permet de fusionner deux listes en comparant les éléments B(I) à B(I + L). Il existe une boucle plus externe pour calculer L (partie entière de N/2, N/4... jusqu'à 0). Cet algorithme est plus rapide que le précédent lorsque les éléments sont rangés de façon aléatoire.

Utilisation de listes chaînes de caractères

Nous allons étudier le problème de la recherche d'un mot dans un dictionnaire pour donner sa traduction. Bien sûr, on suppose que le dictionnaire est limité à une suite de mots rangés avec leur traduction. On a par exemple une suite de mots anglais dans une liste A$ avec leur traduction française dans une liste B$. Lorsque l'on rentre un mot Q$, on le recherche dans le dictionnaire ; s'il y est, on retourne la traduction dans R$ et on l'imprime, sinon on imprime un message "MOT INCONNU". L'algorithme est donc très simple, et il n'est pas nécessaire de le détailler.

```
  1   N = 10
 10   DIM A$ (10), F$ (10)
 20   FOR I = 1 TO N
 30   READ A$ (I), F$ (I)
 40   NEXT I
 50   DATA APPLE, POMME, BASIC, BASIQUE,
           COMMA, VIRGULE, DATA, DONNEE,
           FOR, POUR
 60   DATA IF, SI, LET, SOIT, PRINT, IMPRIMER,
           READ, LIRE, WORD, MOT
 70   INPUT Q$
 80   R$ = "MOT INCONNU"
 90   FOR I = 1 TO N
100   IF Q$ = A$ (I) THEN R$ = F$ (I)
110   NEXT I
```

```
120     PRINT R$
130     GO TO 70
140     END
```

Application à la traduction d'instructions de BASIQUE en BASIC ou inversement

Dans le chapitre précédent nous avions vu comment rechercher si un mot existait dans une chaîne de caractères. Nous venons de voir d'autre part comment traduire un mot. Il suffit donc de fusionner ces deux programmes pour obtenir un programme traducteur de BASIC en BASIQUE, ou inversement.

On suppose que l'instruction BASIQUE est rentrée dans une chaîne appelée BF$. Cette chaîne a pour longueur L1. Il existe une première boucle de balayage des mots clés français et pour chaque mot clé une deuxième boucle de recherche de l'existence de ce mot clé dans la chaîne et son remplacement par le mot clé anglais.

On a donc schématiquement :

```
POUR J = 1 A N
.................................
B$ = MOT CLE FRANÇAIS (J)
.................................
            POUR I = 1 A L1
.........................................................
            RECHERCHE MOT CLE FRANÇAIS (I) DANS LA
            CHAINE BF$ ET REMPLACEMENT PAR MOT
            CLE ANGLAIS (I)
            .............................................
            SUIVANT I
SUIVANT J
```

cela donne le programme suivant :

PROGRAMME DE TRADUCTION BASIQUE-BASIC

```
10      DIM A$ (30), F$ (30)
20      N = 21
30      FOR I = 1 TO N
40      READ A$ (I), F$ (I)
50      NEXT
60      DATA INPUT, ENTRER, NEXT, SUIVANT, THEN,
             ALORS, DATA, DONNEE, FOR, POUR, IF;
             SI
61      DATA LET, SOIT, PRINT, IMPRIMER, GO, ALLER,
             TO, "A", READ, LIRE, LEFT, GAUCHE
```

```
 62   DATA RIGHT, DROITE, SQR, RAC, ON, SUR,
           RETURN, RETOUR, INT, ENT, RESTORE,
           RELIRE
 63   DATA END, FIN, GOSUB, APPEL SOUS PROG,
           LEN, LON
 65   REM ENTREE DE L'INSTRUCTION
 70   INPUT BF$
 80   L1 = LEN (BF$)
 90   FOR J = 1 TO N
 95   G$ = " " : D$ = " "
100   B$ = F$ (J)
110   L2 = LEN (B$)
120   L$ = LEFT (B$, 1)
130   FOR I = 1 TO L1
140   C$ = MID$ (BF$, I, 1)
150   IF C$< >L$ THEN 210
160   M$ = MID$ (BF$, I, L2)
170   IF M$< >B$ THEN 210
175   IF I - 1 = 0 THEN 185
180   G$ = LEFT (BF$, I - 1)
185   IF L1 - I - L2 + 1 = 0 THEN 200
190   D$ = RIGHT$ (BF$, L1 - I - L2 + 1)
200   BF$ = G$ + A$ (J) + D$
205   L3 = LEN (A$(J))
206   L1 = L1 - L2 + L3
207   G$ = " " : D$ = " "
210   NEXT I
220   NEXT J
230   PRINT BF$
240   GO TO 70
```

En fait, ce programme présente un léger défaut, car, s'il fonctionne très bien pour la plupart des instructions, il ne fonctionne pas pour des instructions contenant des caractères, ou :. En effet, dans ce cas l'interpréteur arrête la rentrée de la ligne et ignore le reste des caractères se trouvant après la (,) ou les (:). Pour éviter ce problème, il faut là aussi utiliser l'instruction GET.

Ainsi l'instruction 70 doit être remplacée par la séquence :

```
70   BF$ = " "
71   GET T$
72   IF T$ = " " THEN 71
73   IF ASC (T$) = 13 THEN 80
74   BF$ = BF$ + T$
75   PRINT T$ ;
76   GO TO 71
```

L'instruction 73 teste que l'on a frappé le caractère dont le code ASCII est 13, c'est-à-dire, le caractère RETOUR. Tant que l'on n'a pas obtenu ce caractère on boucle sur l'acceptation des caractères.

Moyennant cette modification, ce programme permet de traduire toute instruction BASIQUE en BASIC. Pour être complet il faudrait l'intégrer au système, interpréteur, ce qui permettrait de disposer d'un interpréteur BASIQUE.

Traitement des tableaux

Le traitement est similaire à celui des listes, mais il faut utiliser deux indices et dimensionner le tableau avec une valeur maximale pour le nombre de lignes et une valeur maximum pour le nombre de colonnes. Comme premier exemple nous prendrons l'impression des tables de multiplication.

```
10    DIM A (10,10)          DIM A (10,10)
20    POUR I = 1 A 10        FOR I = 1 TO 10
30    POUR J = 1 A 10        FOR J = 1 TO 10
40    A (I,J) = I * J        A (I,J) = I * J
50    IMPRIMER A(I,J) ;      PRINT A(I,J) ;
60    SUIVANT J              NEXT J
70    IMPRIMER               PRINT
80    SUIVANT I              NEXT I
90    FIN                    END
```

On obtient en sortie :

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
.......................................
10  20  30  40  50  60  70  80  90 100
```

Autre exemple : Le triangle de PASCAL

On sait que le triangle de PASCAL est obtenu en considérant que les éléments de la première colonne sont égaux à 1 : $a_{11} = 1 \ldots . a_{i1} = 1$, et les autres éléments sont obtenus en considérant la relation :

$$a_{ij} = a_{i-1j} + a_{i-1j-1} \qquad \text{pour } i, j > 1 \text{ et } j \leqslant i.$$

Autrement dit, un élément est égal à la somme de l'élément qui se trouve au-dessus de lui et de l'élément qui se trouve à gauche du précédent. L'on suppose que les éléments non définis sur une ligne sont nuls.

Le programme correspondant est alors :

```
10    DIM A (10,10)
15    A(1,1) = 1
16    PRINT A(1,1)
```

```
20        FOR I = 2 TO 10
25        A(I,1) = 1
26        PRINT A(I,1);
30        FOR J = 2 TO I
40        A(I,J) = A (I - 1, J) + A (I - 1, J - 1)
50        PRINT A(I,J);
60        NEXT J
70        PRINT
80        NEXT I
90        END
```

En l'exécutant on obtient :

```
1
1   1
1   2    1
1   3    3    1
1   4    6    4     1
1   5   10   10     5     1
1   6   15   20    15     6    1
1   7   21   35    35    21    7    1
1   8   28   56    70    56   28    8   1
1   9   36   84   126   126   84   36   9   1
```

2. LE TRAITEMENT DE MATRICES

Dans les BASIC standard il n'existe pas toujours d'instructions de traitement de matrices. Il est cependant possible de considérer des tableaux à deux dimensions comme des matrices et de programmer les opérations habituelles sur des matrices. Dans la suite, nous supposerons que nous avons des matrices carrées.

2-1. Réalisation d'une matrice unité

La matrice unité est caractérisée par les éléments de la diagonale principale égale à l'unité.

Exemple :

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Une telle matrice est caractérisée par des éléments :

$a_{ii} = 1 \quad \forall i$

et $a_{ij} = 0 \quad$ pour $i \neq j$

Une telle matrice peut être engendrée par le programme suivant :

Soit en BASIC :

```
10  DIM A(10,10)                     10  DIM A (10,10)
20  POUR I = 1 A 10                  20  FOR I = 1 TO 10
30  POUR J = 1 A 10                  30  FOR J =  1 TO 10
40  A(I,J) = 0                       40  A(I,J) = 0
50  SI I = J ALORS A(I,J) = 1        50  IF I = J THEN A(I,J) = 1
60  IMPRIMER A(I,J);                 60  PRINT A(I,J);
70  SUIVANT J                        70  NEXT J
80  IMPRIMER                         80  PRINT
90  SUIVANT I                        90  NEXT I
100 FIN                              100 END
```

2-2. Addition de deux matrices

L'addition de deux matrices consiste à ajouter les éléments de même indice. Soit C (I,J) = A(I,J) + B(I,J)

On obtient alors le programme suivant :

```
10  DIM A(3,3), B(3,3), C(3,3)
15  N = 3
16  REM LECTURE DES MATRICES
20  FOR I = 1 TO N
30  FOR J = 1 TO N
40  READ A(I,J), B(I,J)
50  NEXT J, I
60  DATA 1, 2, 3, 4, 2, 1, 0, 3, 5
70  DATA 0, 1, 2, 3, 4, 2, 5, 3, 0
75  REM CALCUL DE LA SOMME
80  FOR I = 1 TO N
90  FOR J = 1 TO N
100 C (I,J) = A(I,J) + B(I,J)
110 NEXT J,I
115 REM IMPRESSION DES MATRICES
120 FOR I = 1 TO N
130 FOR J =  1 TO N
140 PRINT C(I,J);
150 NEXT J
155 IF I = INT (K/2) + 1 THEN PRINT " = "; : GO TO 170
160 PRINT" "
170 FOR J = 1 TO N
180 PRINT A(I,J);
190 NEXT J
200 IF I = INT (K/2) + 1 THEN PRINT " + "; : GO TO 220
```

```
210  PRINT " "
220  FOR J =  1 TO N
230  PRINT B(I,J) ;
240  NEXT J
250  PRINT
260  NEXT I
270  END
```

Remarques. – Dans ce programme, la partie calcul proprement dite est très courte (instructions 80 à 110). Le reste des instructions sont les instructions de lecture des matrices et surtout l'impression ligne à ligne avec un minimum d'édition. C'est ce qu'effectuent les instructions 130 à 260. Le résultat obtenu est alors :

```
3  7  3          1  3  2          2  4  1
3  5  3    =     0  5  1    +     3  0  2
7  7  3          3  2  3          4  5  0
```

Multiplication d'une matrice par un scalaire

Etant donné une matrice A(N,N) on obtient une matrice B(N,N) = K × A(N,N) en multipliant tous les élements de A par une coefficient scalaire K. Autrement dit :

$$b_{ij} = k \times a_{ij}.$$

Le corps de programme permettant d'effectuer ce calcul est :

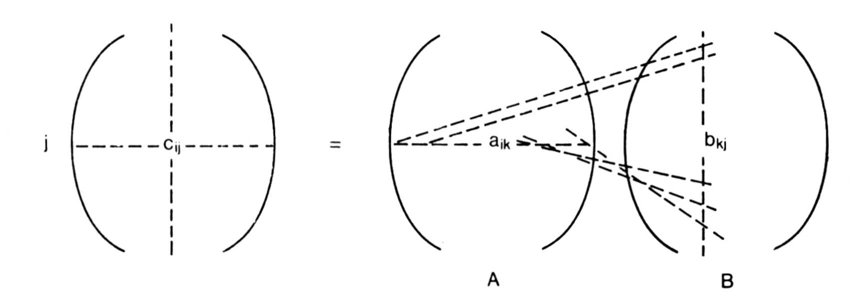
```
FOR   I = 1 TO N
FOR   J = 1 TO N
B(I,J)   = K * A(I,J)
NEXT J,I
```

Application. – Supposons que l'on ait une matrice (ou tableau) représentant les prix unitaires hors taxe de produits vendus par quantité. La multiplication par un facteur *k* égal au % de T.V.A. permet d'obtenir les taxes à payer pour ces produits dans chacun des cas.

2-3. Multiplication de deux matrices

L'opération de multiplication est un peu plus complexe, puisqu'un élément de la matrice produit fait intervenir la somme des produits des éléments de la ligne I et de la colonne J.

$$C_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$



Dans cet exemple, nous ne reprendrons pas la partie lecture et impression des matrices. Seule est modifiée la partie calcul (Instructions 80 à 110). On conserve les deux boucles sur I et J, mais il faut y rajouter une troisième boucle permettant de calculer l'élément c_{ij} d'après la formule donnée ci-dessus. On obtient :

```
FOR I =  1  TO N
FOR J =  1  TO N
C(I,J) =  0
FOR K =  1  TO  N
C(I,J) =    C(I,J) + A(I,K) * B(K,J)
NEXT     K,J,I
```

L'exécution avec les données précédentes donne :

19	14	7		1	3	2		2	4	1
19	5	10	=	0	5	1	×	3	0	2
24	27	7		3	2	3		4	5	0

Remarques. − Il est également possible de multiplier deux matrices rectangulaires à condition que le nombre de colonnes de la première matrice soit égal au nombre de lignes de la deuxième matrice. Ainsi peut-on multiplier une matrice A(L,N) avec une matrice B(N,M), et on obtient une matrice produit C(L,M). Pour cela il suffit de modifier le programme précédent en changeant les bornes des instructions POUR (FOR) :

```
FOR I =  1 TO L
FOR J =  1 TO M
C(I,J) =  0
FOR K =  1 TO N
C(I,J) =    C(I,J) + A(I,K) * B (K,J)
NEXT     K,J,I
```

Application. – Soit une matrice de prix de revient de produits fabriqués à différents prix suivant les quantités : on suppose que les prix de revient par quantités sont en colonnes et que le prix peut être décomposé en différentes parties (matière première, fabrication, vérification, transport). Soit par exemple quatre parties P1,P2,P3,P4, correspondant à trois taux : un taux T1 pour des quantités de 1 à 9, un taux T2 pour des quantités de 10 à 99 et un taux T3 pour des quantités supérieures à 100. On obtient une matrice B :

	P1	P2	P3	P4	
	5	10	15	20	T1
B =	4	8	12	15	T2
	3	6	10	10	T3

Soit, d'autre part, une série de succursales ou de manufactures de fabrication M1,M2,M3,M4,M5 qui ont produit chacune des quantités Q1,Q2,Q3, de ces produits à ces différents taux : On obtient alors une matrice A :

	Q1	Q2	Q3	
	10	20	100	M1
	7	30	200	M2
A =	6	25	150	M3
	5	10	0	M4
	4	11	0	M5

Le problème posé est d'obtenir pour chaque manufacture les prix de revient correspondant aux différents postes ou parties, P1,P2,P3,P4. Pour cela il suffit de calculer pour chaque poste de chaque manufacture le prix de revient global soit :

$$Q1 \times T1 + Q2 \times T2 + Q3 \times T3.$$

Exprimé en notation matricielle, cela revient donc à multiplier la matrice A(5,3) par la matrice B(3,4) et on obtient une matrice C(5,4).

	P1	P2	P3	P4	
	430	860	1390	1500	M1
	755	1510	2465	2590	M2
C =	580	1160	1890	1995	M3
	65	130	195	250	M4
	64	128	192	245	M5

Nous laissons au lecteur le soin d'écrire le programme correspondant et d'obtenir le programme qui permet de remplir la matrice C.

SOLUTION DE L'EXERCICE PRECEDENT

```
 10 DIM B(3,4), A(5,3), C(5,4)
 20 FOR I = 1 TO 5
 30 FOR J = 1 TO 3
 40 READ A(I,J)
 50 NEXT J,I
 60 DATA 10,20,100,7,30,200,6,25,150,5,10,0,4,11,0
 70 FOR I = 1 TO 3
 80 FOR J = 1 TO 4
 90 READ B(I,J)
100 DATA 5,10,15,20,4,8,12,15,3,6,10,10
110 NEXT J,I
120 FOR I = 1 TO 5
130 FOR J = 1 TO 4
140 C(I,J) = 0
150 FOR K = 1 TO 3
160 C(I,J) = C(I,J) + A(I,K) * B(K,J)
170 NEXT K
180 PRINT C(I,J) ;
190 NEXT J
200 PRINT
210 NEXT I
220 END
```

Les instructions spécifiques de manipulation de matrices (MAT)

Dans les exemples ci-dessus nous avons présenté des programmes permettant d'effectuer, à l'aide des instructions de base, des calculs matriciels. Certains BASIC disposent d'instructions permettant d'effectuer directement ces opérations sans avoir à programmer le calcul élément par élément. Comme on vient de le voir, ces instructions ne sont pas nécessaires pour pouvoir effectuer des opérations matricielles, mais elles sont bien pratiques lorsqu'elles existent. Ce sont des instructions ce que l'on regroupe sous le mot clé MAT.

Lecture et écriture des coefficients matriciels

Il est toujours nécessaire de définir une matrice par une déclaration de dimension. Mais pour lire les coefficients d'une matrice A(N,M) il suffit d'écrire :

MAT LIRE A (MAT READ A)

Exemple :

Soit à lire la matrice A de l'exemple précédent.

```
10  DIM B(3,4)
20  MAT LIRE B
30  DONNEE 5,10,15,20,4,8,12,15,3,6,10,10
```

Soit en BASIC

```
10  DIM A(3,4)
20  MAT READ B
30  DATA...
```

De la même façon il est possible d'imprimer une matrice en utilisant une instruction

MAT IMPRIMER B ; (MAT PRINT B ;)

Exemple :

Toujours dans l'exemple précédent, si l'on veut imprimer B. On aura :

```
10  DIM B(3,4)
20  MAT LIRE B                                 MAT READ B
30  DONNEE 5,10,15,20,4,8,12,15,3,6,10,10      DATA ......
40  MAT IMPRIMER B ;                           MAT PRINT B ;
50  FIN
```

Ce qui donnera en sortie :

```
5    10    15    20
4     8    12    15
3     6    10    10
```

Remarque. – Dans certains BASIC il est possible de redimensionner une matrice au moment de la lecture. Ainsi, dans ce cas, les dimensions de la matrice lue sont celles précisées dans l'instruction MAT LIRE. Par exemple l'on peut avoir : MAT LIRE A(3,3), ce qui donnera la lecture d'une matrice 3 lignes 3 colonnes.

Addition et soustraction de matrices

Ces instructions permettent de préciser l'addition ou la soustraction de deux matrices en n'utilisant qu'une seule instruction de type instruction d'affectation.

Le format de l'instruction est :

MAT C = A + B

ou :

MAT C = A – B

ce qui donnera donc une matrice C somme ou soustraction des matrices A et B.

Addition

Cela suppose bien sûr que les *dimensions* des matrices A et B soient *identiques*. On peut également définir des instructions d'addition ou soustraction de type affectation : c'est-à-dire que l'on peut retrouver à droite du signe = le nom de la matrice résultante qui se trouve à gauche du signe =.

Exemple :

MAT A = A + B

ou :

MAT C = C − A

sont des exemples tout à fait corrects.

Dans ce cas bien sûr les contenus initiaux de A et C sont détruits.

Remarques. − L'expression MAT A = B est en général acceptée sur les BASIC disposant d'opérations MAT. Elle permet de spécifier l'égalité de deux matrices.

Initialisation de matrices particulières

Il existe également des instructions permettant d'initialiser les coefficients d'une matrice à des valeurs nulles ou égales à 1.

a) *Mise à zéro d'une matrice*

Les instructions

MAT A = ZER
ou MAT A = ZER(N,M)

permettent d'initialiser les coefficients de la matrice A à zéro. Dans le premier cas la matrice est déjà dimensionnée, dans le deuxième cas on dimensionne en même temps la matrice.

Exemple : MAT A = ZER(2,2) donne

$$A = \begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix}$$

b) *Mise à un d'une matrice*

Les instructions correspondantes sont :

MAT A = CUN (CON)
MAT A = CUN(N,M) (CON(N,M))

Dans le premier cas tous les coefficients de la matrice A sont initialisés à un. Dans le deuxième cas la matrice est dimensionnée en même temps.

Exemple : MAT A = CON (3,3) donne :

$$A = \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$$

c) *Matrice identité :*

Ce sont les instructions

MAT A = IDN

ou : MAT A = IDN(N,N)

On obtient cette fois une matrice qui contient des 1 sur la diagonale principale et des zéros partout ailleurs. Dans tous les cas la matrice identité doit être une matrice carrée à N lignes et N colonnes.

Exemple : MAT A = IDN (4,4) donnera :

$$A = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Multiplication d'une matrice par un scalaire

La forme générale d'une telle instruction est

MAT A = (expression arithmétique) * B

Ceci suppose que les matrices A et B ont mêmes dimensions et que tous les coefficients de B sont multipliés par un facteur qui résulte du calcul de l'expression arithmétique. Cette expression arithmétique peut éventuellement se réduire à une variable numérique ou une constante numérique.

Exemple : MAT A = (X + Y) * B

Dans ce cas, X et Y sont des variables numériques et A et B sont des matrices. Cette instruction aura pour effet de multiplier les coefficients de A par le résultat du calcul de l'expression X + Y.

Multiplication de deux matrices

Si l'on dispose d'instructions MAT, on a également la possibilité de programmer une multiplication de deux matrices en une seule instruction. La forme de l'instruction est alors :

MAT C = A * B

où A,B,C sont des matrices telles que le produit de A par B soit possible, c'est-à-dire par exemple que l'on a des matrices carrées ou rectangulaires telles que A(L,M) et B(M,N) ce qui donne une matrice C(L,N).

Application à l'exemple donné ci-dessus

Le problème que nous avions étudié pour les produits fabriqués à différents taux pour différentes quantités peut maintenant être programmé de la manière suivante :

```
10      DIM A(5,3), B(3,4), C(5,4)
20      MAT LIRE A
30      DONNEE...
40      MAT LIRE B
50      DONNEE...
60      MAT C = A * B
70      IMPRIMER "P1", "P2", "P3", "P4"
80      MAT IMPRIMER C
90      FIN
```

Transposée d'une matrice

La définition de la transposition d'une matrice est l'intervertion des lignes et des colonnes de la matrice. Il existe également une instruction de transposition de matrices :

MAT B = TRN (A)

Exemple :

$$\text{Si : } \quad A = \begin{matrix} 4 & 2 & 3 \\ 5 & 6 & 7 \\ 2 & 1 & 4 \end{matrix}$$

la transposée est :

$$B = A^T = \begin{matrix} 4 & 5 & 2 \\ 2 & 6 & 1 \\ 3 & 7 & 4 \end{matrix}$$

Lorsqu'il s'agit de matrices rectangulaires, les dimensions de la matrice initiale sont (N,M), les dimensions de la transposée sont (M,N).

Inversion d'une matrice

Lorsque l'on considère un produit de matrices carrées $C = A \times B$, si l'on obtient comme matrice C la matrice identité que l'on peut appeler I, on a :

$$I = A \times B$$

On peut alors dire que la matrice B est la matrice inverse de A ($B = A^{-1}$). En effet, comme pour les nombres rationnels, l'inverse d'une matrice X est défini par X^{-1} tel que $XX^{-1} = I$.

Cependant, dans le cas d'une matrice, il se peut que la matrice inverse n'existe pas !

Il faut pour cela qu'elle possède certaines propriétés (son déterminant doit être différent de 0). Ce n'est pas l'objet de cet ouvrage de développer ici les notions mathématiques nécessaires à une formulation générale et précise de ces propriétés. L'objet de ce paragraphe est d'indiquer l'existence d'une instruction MAT permettant de calculer l'inverse d'une matrice lorsqu'elle existe.

Cette instruction a la forme suivante :

MAT B = INV (A)

Application. – Nous nous limiterons à un exemple simple de résolution d'un système d'équations à deux variables :

Soit : $3x + 4y = 10$
$x + 5y = 7$ (1)

Exprimé sous forme matricielle ceci peut s'écrire :

$$\begin{pmatrix} 3 & 4 \\ 1 & 5 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 10 \\ 7 \end{pmatrix}$$

On peut poser :

$$A = \begin{pmatrix} 3 & 4 \\ 1 & 5 \end{pmatrix}$$

$$X = \begin{pmatrix} x \\ y \end{pmatrix} \quad \text{et} \quad B = \begin{pmatrix} 10 \\ 7 \end{pmatrix}$$

Les matrices X et B sont des matrices particulières de dimensions (2,1), que l'on appelle aussi des vecteurs. Le système d'équations (1) peut alors s'écrire sous la forme $A.X = B$.

Pour résoudre ce système d'équations sans l'utilisation de matrices, on peut par exemple utiliser la méthode de substitution. La deuxième équation donne : $x = 7 - 5y$, que l'on substitue dans la première équation soit :

$$3(7 - 5y) + 4y = 10 \text{ d'où } 21 - 15y + 4y = 10$$

ce qui donne $y = 1$ et $x = 2$

En utilisant la notation matricielle, si l'on suppose qu'il existe une matrice inverse A^{-1}, on a :

$$X = A^{-1} B$$

Par définition, cette matrice inverse est telle que :

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} 3 & 4 \\ 1 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

soit : $3\,a_{11} + a_{12} = 1$ $\qquad 3\,a_{21} + a_{22} = 0$

$4\,a_{11} + 5\,a_{12} = 0$ $\qquad 4\,a_{21} + 5\,a_{22} = 1$

d'où $a_{11} = 5/11$ et $a_{21} = -1/11$

$a_{12} = -4/11$ $\qquad a_{22} = +3/11$

on a donc :

$$A^{-1} = \begin{pmatrix} 5/11 & -4/11 \\ -1/11 & 3/11 \end{pmatrix}$$

avec :

$$X = A^{-1} \times B$$

Soit :

$$X = \begin{pmatrix} 5/11 & -4/11 \\ -1/11 & 3/11 \end{pmatrix} \times \begin{pmatrix} 10 \\ 7 \end{pmatrix}$$

on obtient :

$$X = \begin{pmatrix} 50/11 - 28/11 \\ -10/11 + 21/11 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

on trouve donc bien la solution $x = 2$, $y = 1$.

La programmation de ce problème lorsqu'il y a des solutions non dégénérées se résume alors aux instructions suivantes :

```
DIM     A(2,2), B(2,1), X(2,1), A1(2,2)
MAT     READ A
DATA    3,4,1,5
MAT     READ B
DATA    10, 7
MAT     A1 = INV(A)
MAT     X = A1 * B
MAT     PRINT X ;
END
```

Remarque. – Le même programme pourrait être utilisé pour résoudre un système d'équations linéaires avec n inconnues, à condition qu'il y ait une solution.

Exercices

1. *Ecrire un programme de tri alphabétique.*
2. *Ecrire un programme qui permette de mémoriser une partition de musique sous la forme suivante. Les notes seront rentrées en clair, DO, RE, MI, FA, SOL, LA, SI. L'octave sera précisée par un code O suivi d'un chiffre indiquant le numéro de l'octave, les silences par la lettre S. Le rythme sera précisé par un nombre suivant la note : 1 pour une noire, 2 pour une blanche, 4 pour une ronde, 0,5 pour une croche, 0,25 pour une double croche... Les notes pointées sont représentées par la fraction correspondant à la durée de la note. Etablir un tableau correspondant aux fréquences des notes d'une partition.*
3. *Etant donné un entier* N *et N points de coordonnées (X*i, *Y*i*) dans un repère cartésien, écrire un programme BASIC qui détermine la longueur du segment compris entre les points (X*j, *Y*j) *et* (*X*k, *Y*k).
   On demande de lire, N, J, K et le tableau des points.
4. Un carré magique *est un carré divisé en cellules dans lesquelles les nombres entiers à partir de 1 sont arrangés de telle manière que les sommes de chaque ligne, de chaque colonne et de chaque diagonale soient égales.*

Exemple :

4	9	2	
3	5	7	somme = 15
8	1	6	

Il existe plusieurs algorithmes permettant d'obtenir des carrés magiques d'ordre impair (le nombre d'éléments par côté est impair) notamment le plus simple est un algorithme décrit ci-dessous :

– l'élément juste en dessous du centre est rempli par 1 ;

– les éléments suivants : 2, 3, 4... sont remplis dans les cases se trouvant à l'intersection de la ligne du dessous et de la colonne de droite ;

– si l'on arrive au bord du carré, on continue à l'extrémité opposée en suivant la même règle ;

– si une case est déjà remplie, on place le nombre suivant dans la même colonne et deux lignes en dessous.

Ecrire un programme BASIC qui engendre un carré magique en utilisant cet algorithme.

Exemple :

11	24	7	20	3
4	12	25	8	16
17	5	13	21	9
10	18	1	14	22
23	6	19	2	5

5. *Ecrire un programme qui imprime une date en clair à partir de la représentation suivante :*

   n, jj, mm, aa

   où n = *1 à 7 pour LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE*

   jj = *jour dans le mois*
   mm = *numéro du mois*
   aa = *année*
6. *Modifier ce programme pour trouver le jour de la semaine correspondant entre 1900 et 1999. On tiendra compte des années bisextiles et on supposera que le 1er janvier 1900 était un mardi.*

CORRIGES

1. Le principe de l'algorithme de tri est similaire à celui d'un tri sur des nombres, mais ici nous utilisons l'algorithme de tri à « bulles ».
   On compare deux éléments successifs et s'ils ne sont pas dans le bon ordre alors on les échange. Pour ceci il faut effectuer plusieurs « passes » et on s'arrête lorsqu'il n'y a pas eu d'échange au cours d'une passe (c'est la variable E qui indique s'il y a eu des échanges ou pas).
   Le programme est alors :

```
  1  INPUT N
 10  DIM A$(N)
 15  REM LECTURE DES CHAINES DE CARACTERES
 20  FOR I = 1 TO N
 30  INPUT A$(I)
 40  NEXT I
 45  REM DEPUT D'UNE PASSE
 50  E = 0
 60  FOR I = 1 TO N - 1
 70  IF A$ (I)<  = A$(I + 1) THEN 120
 80  C$ = A$(I)
 90  A$(I) = A$ (I + 1)
100  A$ (I + 1) = C$
110  E = 1
120  NEXT I
125  REM ON CONTINUE JUSQU'A CE QU'IL N'Y AIT
     PLUS D'ECHANGES
130  IF E = 1 THEN 50
140  FOR I = 1 TO N
```

```
150 PRINT A$(I)
160 NEXT I
170 END
```

2.
```
 10 REM LECTURE DES NOTES
 20 DIM NO$(12), T(7)
 30 FOR I =  1 TO 12
 40 READ NO$(I)
 50 DATA DO, RE, MI, FA, SOL, LA, SI, O, S."7", "#",
    "b"
 60 NEXT I
 70 REM LECTURE DES INTERVALLES ENTRE NOTES
 80 FOR I = 1 TO 7
 90 READ T(I)
100 DATA 0, 1, 2, 2.5, 3.5, 4.5, 5.5, 6
110 NEXT I
120 REM INITIALISATIONS L = NB DE NOTES
    OC = OCTAVE
130 L = 100
135 DIM N(100), D(100)
140 OC = 1
150 T2 = 0.5
160 F1 = 65
170 REM LECTURE DE LA PARTITION
180 FOR I = 1 TO L
190 INPUT N$(I), D(I)
200 IF N$(I) = "F" THEN L = I - 1 : GO TO 330
210 G$ = LEFT$ (N$(I),2)
220 D$ = RIGHT$ (N$(I),1)
230 FOR J = 1 TO 9
240 IF G$ = NO$(J) THEN 270
250 NEXT J
260 PRINT "ERREUR" : GO TO 190
270 IF G$ = "O" THEN OC = D(I) : GO TO 330
280 IF G$ = "S" THEN F(I) = 0 : GO TO 330
290 F(I) = 6 * OC + T(J)
300 IF D$ = "#" THEN F(I) = F(I) + T2
310 IF D$ = "b" THEN F(I) = F(I) - T2
315 F(I) = F1*2↑ (F(I)/6)
320 NEXT I
330 FOR I = 1 TO L
340 PRINT N$(I) ; "   " ; D(I) ; "   " ; F(I)
350 NEXT I
400 END
```

4. PROGRAMME CARRE MAGIQUE

```
1   REM ENTREE DE LA DIMENSION
5   INPUT N
10  IF N/2 = INT (N/2) THEN 5
20  DIM A (N,N)
30  M = INT (N/2) + 1
35  REM INITIALISATION A0
40  FOR I = 1 TO N
50  FOR J = 1 TO N
60  A(I,J) = 0
70  NEXT J,I
80  K = 1
90  I = M + 1
100 J = M
105 REM CASE EN DESSOUS DU MILIEU = 1
110 A(I,J) = 1
115 REM CASE SUIVANTE
120 K = K + 1
130 IF K > N*N THEN 250
135 REM CAS OU L'ON DEBORDE DU CARRE
140 IF I + 1 > N THEN I = 0
150 IF I + 1 < = 0 THEN I = N - 1
160 IF J + 1 > N THEN J = 0
170 IF J + 1 < = 0 THEN J = N - 1
180 REMCAS OU LA CASE EST DEJA REMPLIE
200 IF A (I + 1, J + 1) = 0 THEN 220
210 I = I + 1 : J = J - 1 : GO TO 140
220 I = I + 1 : J = J + 1
225 REM REMPLIR LA CASE
230 A(I,J) = K
240 GO TO 120
245 REM IMPRESSION DU CARRE
250 FOR I = 1 TO N
260 FOR J = 1 TO N
270 PRINT A (I,J);
280 NEXT J
290 PRINT
300 NEXT I
310 END
```

Exécution

```
? 3 ®
      4   9   2
      3   5   7    La somme des lignes
                   ou des colonnes = 15
      8   1   6
```

```
? 5 ®
        11   24    7   20    3
         4   12   25    8   16
        17    5   13   21    9
        10   18    1    4   22
        23    6   19    2   15
? 8 ®  Les chiffres pairs ne sont pas acceptés.
? 7 ®
        22   47   16   41   10   35    4
         5   23   48   17   42   11   29
        30    6   24   49   18   36   12
        13   31    7   25   43   19   37
        38   14   32    1   26   44   20
        21   39    8   33    2   27   45
        46   15   40    9   34    3   28
```

5.
```
 10  DIM JJ$(7), MM$(12), NJ(12)
 20  FOR I =  1 TO 7
 30  READ JJ$(I)
 40  NEXT I
 50  FOR I = 1 TO 12
 60  READ MM$(I), NJ (I)
 70  NEXT I
 80  DATA LUNDI, MARDI, MERCREDI, JEUDI,
          VENDREDI, SAMEDI, DIMANCHE
 90  DATA  JANVIER, 31, FEVRIER, 29, MARS, 31,
           AVRIL, 30, MAI, 31, JUIN, 30
100  DATA  JUILLET, 31, AOUT, 31 SEPTEMBRE, 30,
           OCTOBRE, 31, NOVEMBRE, 30, DECEMBRE,
           31
110  INPUT N, J, M, A
120  IF N > 7 OR N <  = 0 THEN 110
130  IF M > 12 OR M <  = 0 THEN 110
140  IF A > 99 OR A <  = 0 THEN 110
150  IF J <  = 0 OR J > NJ (M) THEN 110
160  PRINT JJ$(N) ; J ; MM$(M) ; A + 1900
170  END
```

3. LES FONCTIONS ET SOUS-PROGRAMMES

Dans le chapitre 3 nous avons vu l'existence d'un certain nombre de fonctions mathématiques standards que l'on peut utiliser dans des expressions arithmétiques.

Cependant, il peut être nécessaire pour le programmeur de définir de nouvelles fonctions. Ces fonctions seront alors utilisables

dans tout le programme, mais ne pourront être incorporées au langage lui-même. C'est ce que nous allons étudier dans ce chapitre.

3-1. Etude des fonctions standard

Avant de passer à la définition des fonctions particulières, nous allons revenir sur les fonctions mathématiques standard du BASIC.

a) FONCTION VALEUR ABSOLUE

C'est la fonction ABS, avec comme paramètre ou argument une variable numérique positive ou négative.

Y = ABS (X)

permet donc d'obtenir la fonction $Y = |X|$

Exemple :

```
10      ENTRER X
20      Y = ABS (X)
30      IMPRIMER " VALEUR ABSOLUE DE " ; X ; " = " ; Y
40      FIN
```

Exécution :

```
?  10
   □  VALEUR ABSOLUE DE 10 = 10
?  - 7.14
   □  VALEUR ABSOLUE DE - 7.14 = 7.14
```

Cette fonction est particulièrement utile lorsque l'on veut utiliser d'autres fonctions définies uniquement sur les nombres positifs.

b) LA FONCTION ASC

Cette fonction peut être considérée comme une fonction arithmétique dans la mesure où le résultat donne un nombre qui représente le code ASCII en décimal (voir chap. 3) d'un caractère quelconque.

Exemple :

```
10        INPUT C$
20        C = ASC (C$)
30        PRINT "LE CODE ASCII DE" ; C$ ; "EST" ; C
40        GO TO 10
50        END
```

Exécution :

```
?  A ®
   LE CODE ASCII DE A EST 65
?  Z ®
   LE CODE ASCII DE Z EST 90
```

(On peut remarquer que 65 + 25 = 90 et l'on peut vérifier que les codes des lettres de A à Z sont les nombres décimaux de 65 à 90.)

```
?  " " ® représente ici le caractère "blanc"
   LE CODE ASCII DE      EST 32
?  0 ®
   LE CODE ASCII DE 0 EST 48
?  9 ®
   LE CODE ASCII DE 9 EST 57
```

(Ici aussi les codes des chiffres en tant que caractères vont de 48 à 57.)

Avec ce programme on pourrait ainsi obtenir les codes de tous les caractères ASCII. Ces codes sont compris entre 0 et 255

Remarque. – Sur certaines machines il peut y avoir des codes non accessibles, car le clavier ne permet pas toujours d'obtenir tous les caractères possibles.

c) LA FONCTION RACINE CARREE RAC (SQR)

Cette fonction a déjà été utilisée dans différents exercices. Il faut simplement faire attention à ce que l'expression sur laquelle porte la fonction soit positive (cf. ci-dessus). En BASIC anglais son nom est SQR (pour "square root" : racine carrée), en BASIQUE français on l'appellera RAC.

Application : Résolution d'une équation du second degré

Soit : $ax^2 + bx + c = 0$

On sait que si $a = 0$ on est ramené à une équation du premier degré. Dans ce cas, si b est nul et c également, on a une indétermination ; si $c \neq 0$ on a une impossibilité. Dans le cas général il y a des racines réelles si le discriminant $b^2 - 4\,ac$ *est* $\geqslant 0$. Dans ce cas, les racines sont :

$$x_1, x_2 = (-b \pm \sqrt{b^2 - 4\,ac})/2a$$

```
10  ENTRER A,B,C
20  SI A = 0 ALORS 100
30  D = B↑2 - 4 * A * C
40  SI D < 0 ALORS 80
50  X1 = (- B + RAC (D))/(2 * A)
```

```
 60  X2 = ( - B - RAC(D))/2/A
 65  IMPRIMER "X1  = "; X1, "X2 =  "; X2
 70  ALLER A 150
 80  IMPRIMER "PAS DE RACINES REELLES"
 90  ALLER A 150
100  SI B = 0 ALORS 130
110  IMPRIMER "PREMIER DEGRE X = "; - C/B
120  ALLER A 150
130  SI C = 0 ALORS IMPRIMER "INDETERMINE" : ALLER
     A 150
140  IMPRIMER "IMPOSSIBLE"
150  FIN
```

Soit en BASIC :

```
 10  INPUT A,B,C
 20  IF A = 0 THEN 100
 30  D = B ↑ 2 - 4 * A * C
 40  IF D < 0 THEN 80
 50  X1 = ( - B + SQR(D))/2/A
 60  X2 = ( - B - SQR(D))/(2 * A)
 65  PRINT "X1 ="; X1 ; "X2 ="; X2
 70  GO TO 150
 80  PRINT "PAS DE RACINES REELLES"
 90  GO TO 150
100  IF B = 0 THEN 130
110  PRINT "PREMIER DEGRE X ="; - C/B
120  GO TO 150
130  IF C = 0 THEN PRINT "INDETERMINE" : GO TO 150
140  PRINT "IMPOSSIBLE"
150  END
```

Exécution :

?	0, 0, 2 IMPOSSIBLE	équation 2 = 0
?	0, 0, 0 INDETERMINE	équation 0 = 0
?	1, 1, 1	équation $x^2 + x + 1 = 0$
	PAS DE RACINES REELLES	
?	0, 2, 3 PREMIER DEGRE X = - 1.5	équation $2x + 3 = 0$
?	1, - 1, - 2 X1 = 2 X2 = - 1	équation $x^2 - x - 2 = 0$
?	1, - 6, 9 X1 = 3.00006104	équation $x^2 - 6x + 9 = 0$ X2 = 2.99993896

On voit ici également un problème d'arrondi ; l'on aurait dû obtenir une racine double $x = 3$. En fait, si l'on avait calculé D avec l'instruction

D = B * B – 4 * A * C

l'on aurait obtenu X1 = 3   X2 = 3, ce qui montre que l'opération d'exponentiation peut introduire des erreurs d'arrondis.

d) LES FONCTIONS TRIGONOMETRIQUES (ATN, COS, SIN, TAN)

Les fonctions trigonométriques standard sont Arc tangente (ATN), Cosinus (COS), Sinus (SIN) et Tangente (TAN). On les utilise en précisant un paramètre entre parenthèses. Le paramètre représentant la variable ou l'expression entre parenthèses est exprimé en *radians* pour les fonctions COS, SIN et TAN et le résultat de ATN est *en radians*. Le paramètre de ces fonctions peut s'exprimer sous forme d'expressions arithmétiques quelconques dont le résultat est un nombre exprimant une valeur en radians.

Exemples :

– PRINT ATN (1) ®
0.785398163

ou encore :

– PRINT ATN (1) * 4 ®
3. 14159266

ce qui donne bien Arc tangente (1) = $\pi/4$.

– PRINT COS (0) ®
1

Certains claviers disposent même du caractère π, ce qui permet d'obtenir directement des valeurs en fonction de π.

– PRINT COS (π/3) ®
0.5

– PRINT SIN (π/6) ®
.5

– PRINT TAN (π/4) ®
.99999999

Dans ce cas le résultat théorique est 1, et l'on obtient donc un résultat approché par défaut. Ceci est dû aux arrondis dans les calculs de ces fonctions.

Remarque. – Si l'on ne dispose pas du caractère π sur le clavier, il faut définir une variable :

10 PI = 3.1459265

et l'utiliser ensuite dans les fonctions trigonométriques

Exemple :

```
20  PRINT SIN (PI/2)
30  END
```

Exécution :

```
□ .99999652
```

Là aussi, le résultat est approché, car la valeur de π est approchée, alors que l'instruction PRINT SIN (π/2) ® donnerait : □ 1.

On peut également utiliser des multiples de π et obtenir un résultat correct.

Exemple :

PRINT COS (2 * π) ® donne : □ 1.

Quelques problèmes de trigonométrie : conversion de radians en degrés et inversement

On sait que $\pi = 180°$

Si l'on donne donc une valeur d'angle en degrés (D), sa valeur R en radians est égale à :

$$R = \frac{D}{180} \times \pi$$

Inversement, si l'on connaît la valeur en radians, on a donc :

$$D = \frac{R \times 180}{\pi}$$

Soit à imprimer les valeurs des fonctions trigonométriques pour des angles donnés en degrés. On obtient :

```
10      INPUT D
20      R = D * π /180
25      PRINT "COSINUS", "SINUS", "TANGENTE"
30      PRINT COS (R) ; SIN (R) ; TAN (R)
40      GO TO 10
```

L'exécution donnera :

```
?   0            ®
□   COSINUS          SINUS            TANGENTE
    1                0                0
?   30           ®
□   COSINUS          SINUS            TANGENTE
    0.86602504       0.5              0.577350269
?   45           ®
□   COSINUS          SINUS            TANGENTE
    0.707106782      0.707106781      0.9999999
```

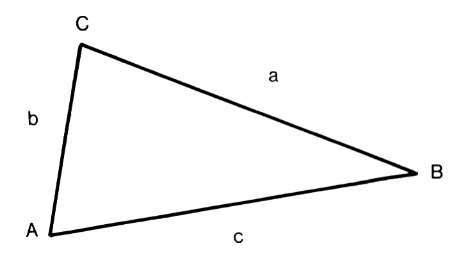
?	60 ®		
□	COSINUS	SINUS	TANGENTE
	0.5	0.866025404	1.7320508
?	90 ®		
□	COSINUS	SINUS	TANGENTE
	0	1	ERREUR DIVISION PAR 0

La dernière exécution donne bien sûr une erreur, car tang (90°) est égale à l' ∞, ce qui se traduit ici par une erreur de division par 0 car cos (90) = 0 et donc :

$$tgx = \frac{\sin x}{\cos x} = \infty \text{ pour } x = 90°.$$

Calcul des angles d'un triangle

Soit un triangle ABC dont les côtés ont pour longueurs *a,b,c.*



correspondant aux angles A, B, C.

En appliquant le théorème de Pythagore généralisé, on sait que :

$$a^2 = b^2 + c^2 - 2\ bc \cos A$$

soit :

$$\cos A = \frac{b^2 + c^2 - a^2}{2\ bc}.$$

Ne disposant pas de la fraction ARC cosinus (A), on est obligé d'utiliser quelques relations trigonométriques simples notamment :

$$\sin^2 A + \cos^2 A = 1, \text{ soit } \sin A = \sqrt{1 - \cos^2 A}$$

$$\text{et } tg\ A = \frac{\sin A}{\cos A} = \frac{\sqrt{1 - \cos^2 A}}{\cos A}$$

d'où :

$$A = \text{Arc tangente} \left(\frac{\sqrt{1 - \cos^2 A}}{\cos A} \right).$$

De même, à partir de :

$$b^2 = a^2 + c^2 - 2\,ac \cos B$$

on aurait :

$$\cos B = \frac{a^2 + c^2 - b^2}{2\,ac}$$

et :

$$B = \textit{Arc tangente} \left(\frac{\sqrt{1 - \cos^2 B}}{\cos B} \right)$$

Quant à C, on l'obtient à l'aide de la relation

$$A + B + C = 180^o \text{ (degrés)}.$$

Remarque. – Le problème n'a pas toujours de solutions en particulier si l'on donne à *a, b, c* des valeurs quelconques.

En effet, si :

$$\frac{b^2 + c^2 - a^2}{2\,bc} > 1$$

il n'y a pas d'angle A réel, ce qui veut dire que dans ce cas il n'y a pas de triangle correspondant aux valeurs *a, b, c.* Le programme correspondant est donné ci-dessous :

```
10  INPUT A,B,C
20  CA = (B↑2 + C↑2 − A↑2) / (2 * B * C)
30  CB = (A↑2 + C↑2 − B↑2) / (2 * A * C)
40  SA = SQR (1 − CA↑2)
50  SB = SQR (1 − CB↑2)
60  AA = ATN (SA/CA) * 180 / π
70  BB = ATN (SB/CB) * 180 / π
80  CC = 180 − AA − BB
90  PRINT "A ="; A; "B ="; B; "C ="; C
100 PRINT "ANGLE A = "; AA; "ANGLE B = "; BB;
    "ANGLE C = "; CC
110 END
```

L'exécution de ce programme donne :

```
?   2, 3, 4 ®
□   A = 2          B = 3          C = 4
    ANGLE A = 28.9550243  ANGLE B = 46.5674635
    ANGLE C = 104.477512
?   3, 4, 5 ®
□   A = 3          B = 4          C = 5
    ANGLE A = 36.8698977  ANGLE B = 53.1301023
    ANGLE C = 90
```

(en effet, il s'agit d'un triangle rectangle puisque $A^2 + B^2 = C^2$)

```
?   1, 1, 1.414 ®
□   A = 1          B = 1          C = 1.414
    ANGLE A = 45.0086517  ANGLE B = 45.0086519
    ANGLE C = 89.9826968
```

(il s'agit cette fois d'un triangle rectangle isocèle puisque $1{,}414 \simeq \sqrt{2}$)

```
?   1, 2, 3, ®
□   ERREUR SUR L'INSTRUCTION 40 (ILLEGAL QUANTITY ERROR IN 40)
```

Dans ce cas :

$$\frac{B^2 + C^2 - A^2}{2\,BC} = \frac{4 + 9 - 1}{2 \times 2 \times 3} = \frac{12}{12} = 1$$

Soit cos A = 1 d'où A = 0, ce qui montre qu'il n'y a pas de triangle dans ce cas.

Cependant lors de l'exécution, l'erreur n'aurait pas dû se produire sur l'instruction 40, puisque :

$$CA = 1 \text{ et que } SA = \sqrt{1 - CA^2} = 0.$$

En effet, si l'on imprime $1 - CA^2$ avant d'effectuer l'instruction 40, on trouve : − 2.25918484 E − 09, c'est-à-dire une valeur très voisine de 0, mais négative ! Ceci illustre les problèmes d'arrondis que l'on peut rencontrer dans certains programmes, en particulier lorsque les valeurs obtenues sont très voisines d'une valeur critique !

e) LES FONCTIONS EXPONENTIELLES ET LOGARITHME

Les fonctions mathématiques correspondantes sont e^x et logarithme népérien $\log_e(x)$. Nous en avons déjà vu des exemples. Les formes générales sont EXP (expression arithmétique) et LOG (expression).

Exemple :

PRINT EXP (1) ®
□ 2, 71828183
PRINT LOG (1) ®
□ 0
PRINT LOG (EXP(1)) ®
□ 1

On peut utiliser comme paramètres n'importe quelle expression arithmétique avec la seule contrainte que pour le LOG la valeur de l'expression doit être positive. On peut en particulier définir les fonctions hyperboliques sin *h*, cos *h* et tg *h* en utilisant la fonction exponentielle :

SH = (EXP(X) − EXP (− X))/2
CH = (EXP(X) + EXP (− X))/2

$$TH = \frac{SH}{CH}$$

Nous verrons plus loin qu'il est possible de définir ces fonctions une fois pour toutes de manière symbolique dans un programme.

Fonction exponentielle de base quelconque

La fonction exponentielle peut également servir à calculer des fonctions de type a^x en utilisant la correspondance :

$$a^x = e^{x \text{Log}\, a}$$

Ceci peut d'ailleurs aussi être programmé en utilisant l'opérateur d'exponentiation (a^x est équivalent à A↑X).

Exemples :

```
10      INPUT A,X
20      PRINT A↑X, EXP (X * LOG(A))
30      GO TO 10
```

Exécution

```
?       2, 2 ®
□       4            4           (2² = e² Log2)
?       2, 4.5 ®
□       22.627417         22.627417(2⁴·⁵ = e⁴·⁵Log2)
```

On voit donc que :

EXP (expression) est équivalent à *e*↑ (expression)

Fonction logarithmique de base quelconque

La fonction logarithme népérien est la fonction inverse de l'exponentielle, puisque par définition :

$$\text{Log } e^x = x \text{ et } e^{\text{Log } x} = x$$

Si l'on considère un logarithme de base a soit : $\text{Log}_a(x)$ on a :

$$\text{Log}_a(x) = \text{Log}_a(e^{\text{Log } x})$$
$$\text{Log}_a(x) = \text{Log}_a(e) \times \text{Log}(x)$$

En particulier, si $x = a$ on obtient

$$\text{Log}_a(a) = 1 = \text{Log}_a(e) \times \text{Log}(a)$$
$$\text{Log}(a) = \frac{1}{\text{Log}_a(e)}$$

On a donc :

$$\text{Log}_a(x) = \frac{\text{Log}(x)}{\text{Log}(a)}$$

Exemple :

```
10      INPUT A,X
20      LA = LOG(X) / LOG(A)
30      PRINT "LOG A BASE" ; A ; "DE" ; X ; " =" ; LA
40      GO TO 10
```

Exécution :

```
?    10, 100 ®
     LOG A BASE 10 DE 100 = 2
?    2, 8 ®
     LOG A BASE 2 DE 8 = 3
?    10, 45 ®
     LOG A BASE 10 DE 45 = 1.6532151
?    10, 3 ®
     LOG A BASE 10 DE 3 = 0.477121255
```

f) LES FONCTIONS PARTIE ENTIERE ET SIGNE (INT, SGN)

La fonction ENT (INT) permet d'obtenir la partie entière d'une valeur décimale ou fractionnaire.

La fonction SGN permet d'obtenir le signe d'une expression. Le signe est représenté par une valeur − 1 si ce signe est négatif, + 1 s'il est positif et 0 si la valeur est nulle.

Exemple :

```
10 ENTRER X                        10 INPUT X
20 IMPRIMER ENT (X);
   SGN(X)                          20 PRINT INT(X); SGN(X)
30 ALLER A 10                      30 GO TO 10
```

Exécution :

```
?  0 ®
□  0        0
?  - 0 ®
□  0        0
?  + 0 ®
□  0        0
?  3.5 ®
□  3        1
?  - 6.2 ®
□  - 7      - 1
?  - 0.25 ®
□  - 1      - 1
?  + 0.56 ®
□  0        1
```

Remarque. – La partie entière d'un nombre négatif correspond au nombre entier négatif immédiatement inférieur (cf. les exemples ci-dessus).

Exemple d'application

Déterminer si un nombre entier N est premier. Pour cela il suffit de tester si tous les entiers à partir de 2 jusqu'à la valeur de la racine carrée de ce nombre $\sqrt{N}$ ne sont pas diviseurs de ce nombre. En effet, si l'on n'a pas trouvé de nombres divisant N avant $\sqrt{N}$, les nombres au-delà de $\sqrt{N}$ seront nécessairement associés à un facteur qui aurait déjà dû être trouvé comme diviseur de N puisque $\sqrt{N} \times \sqrt{N} = N$. Donc, si l'on prend un nombre $X > \sqrt{N}$, alors l'autre facteur Y doit être $< \sqrt{N}$.

Pour déterminer si un nombre est premier, il suffit de tester si la partie entière de la division de N par tous les nombres de 2 à $\sqrt{N}$ est égale à cette division.

On obtient alors le programme suivant :

```
10      ENTRER N
20      POUR I = 2 A RAC(N)
30      SI N/I = ENT(N/I) ALORS 60
40      SUIVANT I
50      IMPRIMER N ; "EST PREMIER" : ALLER A 10
```

```
60        IMPRIMER N ; "EST DIVISIBLE PAR" ; I
70        ALLER A 10
80        FIN
```

En BASIC, on obtient :

```
10        INPUT N
20        FOR I = 2 TO SQR(N)
30        IF N/I = INT(N/I) THEN 60
40        NEXT I
50        PRINT N ; "EST PREMIER" : GO TO 10
60        PRINT N ; "EST DIVISIBLE PAR" ; I
70        GO TO 10
80        END
```

Exécution :

```
?     11          ®
      11 EST PREMIER
?     39          ®
      39 EST DIVISIBLE PAR 3
?     37          ®
      37 EST PREMIER
?     1 789       ®
      1 789 EST PREMIER
?     10 789      ®
      10 789 EST PREMIER
```

Autre exercice

Modifier ce programme pour obtenir la liste des nombres premiers jusqu'à 1 000.

Pour cela, il suffit de rajouter une boucle plus externe au programme précédent.

```
10        FOR N = 1 TO 1000
20        FOR I = 2 TO SQR(N)
30        IF N/I = INT(N/I) THEN 60
40        NEXT I
50        PRINT N ;
60        NEXT N
70        END
```

Exécution :

```
 1   2   3   5   7   11   13   17   19   23  29
31  37  41  43  47   53   59   61   67   71
73  79  83  89  97  101  103  107  109  113 ...
```

Exercices

1. *Un nombre parfait est un nombre qui est caractérisé par le fait qu'il est égal à la somme de tous ses diviseurs excepté lui-même. Le premier nombre parfait est 6 qui est égal à 1 + 2 + 3 qui sont bien les diviseurs de 6.*
   Ecrire un programme qui édite la liste de tous les nombres parfaits jusqu'à 200.

2. *Deux nombres* m *et* n *sont appelés nombres amis si la somme des diviseurs de* m *est égale à* n *et la somme de tous les diviseurs de* n *est égale à* m.
   Exemple : 220 et 284 sont des nombres amis.
   Ecrire un programme qui permet de trouver au moins une autre paire de nombres amis.

   Remarque. – *Pour trouver la prochaine paire de nombres amis prévoir d'aller jusqu'à 2 000.*

3. *Parmi tous les entiers > 1, il y en a seulement quatre qui peuvent être représentés par la somme des cubes de leurs chiffres.*
   Ainsi par exemple : $153 = 1^3 + 5^3 + 3^3$
   Ecrire un programme pour déterminer les trois autres.
   Note. – *Les quatre nombres sont compris entre 150 et 410.*

4. *La « preuve perdue »* de *FERMAT :*
   Le mathématicien FERMAT a prétendu avoir démontré un théorème dont on n'a jamais retrouvé la preuve et que l'on n'a jamais pu infirmer.

   Il s'agit de prouver que l'on ne peut pas trouver des valeurs entières a, b, c, *telles que* $a^n + b^n = c^n$ *pour* $n > 2$

   1° *Afin de le vérifier sur un sous-ensemble des nombres entiers, on demande d'écrire un programme qui permette de faire varier* a, b, c *entre 1 et 30 pour toutes les valeurs de* n *comprises entre 3 et 5.*

   2° *On déterminera d'autre part tous les triplets* a, b, c *tels que* $1 \leqslant a < b < c$ *pour lesquels :*

$$|a^n + b^n - c^n| \leqslant 15$$

CORRIGES

1. *Programme nombres parfaits :*

```
 10     INPUT N
 20     FOR I = 2 TO N
 30     S = 1
 40     M = I/2
 50     FOR J = 2 TO M
 60     IF I/J = INT (I/J) THEN S = S + J
 70     NEXT J
 80     IF S = I THEN PRINT I ; "EST PARFAIT"
 80     NEXT I
100     END
```

Exécution :

```
?      30 ®
□       6 EST PARFAIT
       28 EST PARFAIT
```

2. *Nombres amis*

```
 10     INPUT N
 20     FOR I = 2 TO N
 30     S = 1
 40     M = I/2
 50     FOR J = 2 TO M
 60     IF I/J = INT (I/J) THEN S = S + J
 70     NEXT J
 80     IF S = I THEN 180
 90     IF S>I THEN 200
100     M = S/2
110     NS = 1
120     FOR J = 2 TO M
130     IF S/J = INT(S/J) THEN NS = NS + J
140     NEXT J
150     IF NS< >I THEN 200
160     PRINT I ; S ; "SONT DES NOMBRES AMIS"
170     GO TO 200
180     PRINT I ; "EST PARFAIT"
200     NEXT I
210     END
```

Exécution :

```
?      300 ®
□        6 EST PARFAIT
        28 EST PARFAIT
       284 220 SONT DES NOMBRES AMIS.
```

3. *Programme*

```
10    FOR I = 1 TO 9
20    FOR J = 0 TO 9
30    FOR K = 0 TO 9
40    N = I * 10 * 10 + J * 10 + K
50    N3 = I↑3 + J↑3 + K↑3
60    IF INT (N3) = INT (N) THEN PRINT N3 ; N
70    NEXT K, J, I
80    END
```

Exécution :

```
153    153
370    370
371    371
407    407
```

Autre solution :

```
10    FOR I = 1 TO 9
20    FOR J = 0 TO 9
30    FOR K = 0 TO 9
40    N = I * 10 * 10 + J * 10 + K
50    N3 = I * I * I + J * J * J + K * K * K
60    IF N3 = N THEN PRINT N3 ; " = " ; I ;
      "↑3 + " ; J ; "↑3 + " ; K ; "↑3"
70    NEXT K, J, I
80    END
```

Exécution :

```
153 = 1↑3 + 5↑3 + 3↑3
370 = 3↑3 + 7↑3 + 0↑3
371 = 3↑3 + 7↑3 + 1↑3
407 = 4↑3 + 0↑3 + 7↑3
```

La fonction génératrice de nombres aléatoires

Cette fonction, appelée ALE en BASIQUE ou RND (Random) en BASIC, permet de donner un nombre tiré au hasard et compris entre 0 et 1. En principe cette fonction doit avoir un paramètre entre parenthèses, mais dans certains BASIC, il suffit de préciser RND.

Exemple :

```
1    FOR I = 1 TO 10
2    PRINT RND (1)
3    NEXT I
4    END
```

Exécution :

```
0.285986470
0.332769204
0.800480362
0.616685484
0.652093045
0.187796171
0.309893282
0.510722272
0.869524463
0.206369258
```

Remarques. – Sur certains BASIC, si l'on réexécute le même programme, la suite de nombres est identique ; ceci nécessitera alors l'utilisation d'une instruction spéciale RANDOMIZE qui fait varier les séries de chiffres tirés au hasard.

– Sur la plupart des BASIC réalisés sur micro-ordinateurs cela n'est pas nécessaire et à chaque exécution la suite est modifiée si l'on utilise un paramètre négatif RND (– 1)...

– Si l'on avait utilisé RND (I) au lieu de RND (1), on aurait également obtenu une suite de nombres au hasard.

Application : Simulation d'un jeu de dés

Le lancement d'un dé est un phénomène aléatoire qui peut prendre six valeurs : 1,2,3,4,5,6. Pour obtenir une simulation d'un tel jeu, il suffit donc de multiplier le résultat de la fonction RND par 6 et de rajouter 1 de façon à obtenir des valeurs allant de 1, ...à 6.999..., dont la partie entière est bien un nombre de 1 à 6.

Dans le programme suivant, nous allons jouer cinq séries de 24 coups et nous compterons le nombre de fois que chaque face est sortie.

```
  1    DIM D(6)
 10    FOR I = 1 TO 5
 15    REM INITIALISER LE NOMBRE DE FIGURES
 20    FOR J = 1 TO 6
 30    D(J) = 0
 40    NEXT J
 45    REM LANCEMENT DU DE (24 FOIS)
 50    FOR K = 1 TO 24
 60    DE = INT (6 * RND(1) + 1)
 70    PRINT DE ;
 80    D(DE) = D(DE) + 1
 90    NEXT K
100    PRINT
105    REM IMPRIMER LES RESULTATS
```

```
110     FOR J = 1 TO 6
120     PRINT D(J);
130     NEXT J
140     PRINT : PRINT
150     NEXT I
160     END
```

Exécution :

```
5  2  2  1  5  5  6  3  5  4  2  3  4
2  4  1  5  5  4  4  5  5  6  1
3  4  2  5  8  2   : les 6 derniers chiffres représentent le
                     nombre de fois que chaque face est sortie
                     (3 fois 1, 4 fois 2, 2 fois 3, 5 fois 4,
                     8 fois 5, 2 fois 6)
2  5  6  4  4  5  1  6  6  1  3  6  6
3  6  4  6  2  3  2  1  3  1  2
4  4  4  3  2  7
6  2  2  6  1  1  4  6  2  1  4  3  6
2  4  1  4  2  3  3  3  1  5  5
5  5  4  4  2  4
2  1  6  5  3  2  6  6  1  2  1  4  2
4  2  3  1  1  4  6  1  3  2  6
6  6  3  3  1  5
5  6  2  6  5  5  6  5  4  5  2  2  1
2  1  3  6  1  6  5  2  4  2  4
3  6  1  3  6  5
```

Génération d'un nombre aléatoire compris entre deux valeurs A et B (B > A)

L'intervalle de variation est B − A ; la valeur initiale étant A, il suffit de définir

N = A + (B − A) * RDN(1)

Application au Jeu de LOTO

Les numéros qui peuvent sortir vont de 1 à 49, et il y a sept numéros qui sont sélectionnés. On a donc :

```
10      FOR I = 1 TO 7
20      N = INT (49 * RND(1) + 1)
30      PRINT N ;
40      NEXT I
50      END
```

Exécution :

2	8	1	22	15	23	46
41	10	40	12	14	44	22
20	9	47	7	31	35	42
42	29	24	22	41	19	7
38	35	7	33	9	44	16

Remarque. – Ce programme n'est pas complet, car il n'exclut pas le tirage de deux nombres identiques. Dans chacune des listes nous n'avons pas eu de duplication du même nombre, ce qui est dû à ce que nous avons eu de la chance !

Exercices

1. *Ecrire un programme qui joue à pile ou face. Imprimer le résultat sous forme d'une suite de P ou F en comptant le nombre de Pile ou de Face pour un nombre de coups que l'on considère comme une donnée.*
2. *Ecrire un programme qui permet de sortir une liste de nombres au hasard en s'assurant que l'on n'obtient pas deux fois le même nombre.*
3. *Appliquer l'exercice 2 à la modification du jeu de LOTO.*

Corrigé « Pile ou face »

```
 10     INPUT N
 20     F = 0 : P = 0
 30     FOR I = 1 TO N
 40     C = INT (2 * RND(I))
 50     IF C = 1 THEN 80
 60     PRINT "P" ;
 65     P = P + 1
 70     GO TO 100
 80     PRINT "F" ;
 90     F = F + 1
100     NEXT I
110     PRINT
120     PRINT P ; "PILES" ; F ; "FACES"
130     GO TO 10
```

Exécution :

```
?    10  ®
     FFPPFFPFFP
     4 PILES      6 FACES
?    20  ®
     PFPPPFFPPPFFPFPPFPPP
     13 PILES     7 FACES
?    10  ®
     FFPPPPFPFF
     5 PILES      5 FACES
?    20   ®
     PFPFPFFPPPPPPPFPPFFF
     12 PILES     8 FACES
```

On voit donc sur ces exemples que les séquences varient d'une exécution à la suivante.

Exercices

1. *Ecrire un programme de conversion décimal binaire pour des nombres ayant au plus 10 chiffres binaires (bits) (⩽1023)*
2. *Ecrire un programme de conversion d'un nombre décimal en une base quelconque ⩽10.*
3. *Ecrire un programme de conversion de décimal en hexadécimal (base 16). Dans ce cas les nombres de 10 à 15 sont représentés par les lettres A,B,C,D,E,F (chiffres hexadécimaux).*
4. *Ecrire un programme de conversion d'un nombre d'une base quelconque (⩽16) en un nombre décimal.*

Corrigés des exercices

1. *Conversion décimale binaire*

```
10      INPUT N
15      PRINT N ; "EN BINAIRE = "
20      FOR I = 10 TO 0 STEP - 1
30      B = INT (N/2↑I)
40      N = N - B * 2↑I
50      PRINT B ;
60      NEXT I
70      END
```

Exécution :

```
?       11 ®
□       11 EN BINAIRE = 0000001011
?       77 ®
□       77 EN BINAIRE = 0001001101
```

2. *Conversion décimale en base quelconque ⩽10*

```
 1      INPUT "BASE" ; B
 2      IF B > 10 OR B < 2 THEN 1
10      INPUT "NOMBRE" ; N
15      PRINT N ; "EN BASE" ; B ; " = " ;
20      FOR I = 10 TO 0 STEP - 1
25      D = INT (B↑I)
30      C = INT (N/D)
40      N = N - C * D
50      PRINT C ;
60      NEXT I
70      PRINT
80      GO TO 10
```

Remarque. – Le passage par la variable D est nécessaire pour éviter des erreurs d'arrondis qui peuvent donner des résultats faux sur certaines machines.

Exécution :

```
        BASE ? 10 ®
        NOMBRE ? 89 ®
□       89 EN BASE 10 = 00000000089
        NOMBRE ? 1979 ®
□       1979 EN BASE 10 = 00000001979
        BASE ? 8 ®
        NOMBRE ? 45 ®
□       45 EN BASE 8 = 00000000055
        NOMBRE ? 1979 ®
□       1979 EN BASE 8 = 00000003673
        BASE ? 5 ®
        NOMBRE 45 ®
□       45 EN BASE 5 = 00000000140
        BASE ? 3 ®
        NOMBRE 45 ®
□       45 EN BASE 3 = 0000001200
```

3. *Programme de conversion d'un nombre décimal en un nombre dans une base quelconque*

```
 1      INPUT "BASE" ; B
 2      IF B > 16 OR B < 2 THEN 1
 3      DIM H$ (16)
```

```
4       FOR I =  0 TO 15
5       READ H$ (I)
6       NEXT I
7       DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,
10      INPUT "NOMBRE" ; N
15      PRINT N ; "EN BASE" ; B ; " = " ;
20      FOR I = 10 TO 0 STEP  - 1
25      D = INT (B↑I)
30      C = INT (N/D)
40      N = N - C * D
50      PRINT H $ (C) ;
60      NEXT I
65      PRINT
70      GO TO 10
```

Exécution :

```
        BASE ? 16 ®
        NOMBRE ? 45 ®
   □    45 EN BASE 16 = 000000002D
        NOMBRE ? 89 ®
   □    89 EN BASE 16 = 000000059
```

Remarque. – Ce programme est général pour n'importe quelle base ⩽16. Il peut donc remplacer avantageusement les deux programmes précédents.

4. *Programme de conversion d'un nombre en base quelconque en décimal*

```
 10     INPUT "BASE" ; B
 20     DIM A$ (16)
 30     IF B > 16 OR B < 2 THEN 10
 40     FOR I = 0 TO 15
 50     READ A$ (I)
 60     NEXT I
 70     DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
 80     INPUT "NOMBRE" ; N$
 90     PRINT N$ ; "EN BASE" ; B ; " = "
100     K = LEN (N$)
110     M = 0
120     FOR I = 1 TO K
130     L$ = RIGHT$ (N$,1)
140     FOR J = 0 TO B - 1
150     IF L$ = A$ (J) THEN 175
160     NEXT J
170     GO TO 80
175     IF K - I = 0 THEN 190
180     N$ = LEFT$ (N$, K - I)
```

```
190     M = M + J * B↑ (I − 1)
200     NEXT I
210     PRINT M ; "EN DECIMAL"
215     PRINT
220     GO TO 80
230     END
```

Exécution :

```
        BASE ? 7 ®
        NOMBRE ? 145 ®
   □    145 EN BASE 7 =  82 EN DECIMAL
        BASE ? 16 ®
        NOMBRE ? 1 AC ®
   □    1AC EN BASE 16 = 428 EN DECIMAL
```

3-2. Les fonctions définies par le programmeur

Lorsque l'on désire définir des fonctions qui ne font pas partie de la liste standard étudiée ci-dessus, il faut utiliser une instruction de *déclaration* commençant par le mot clé DEF (Définition). Cette fonction sera alors utilisable dans toute la suite du programme.

FORME GENERALE DE LA DECLARATION DE FONCTION

Cette déclaration a la forme suivante :

NN DEF FNK(X) = Expression

– NN est un numéro de ligne.

– FNK est un nom de fonction défini par le programmeur : il doit commencer par FN et être suivi par un nom d'identificateur BASIC, mais ne doit pas être un mot réservé du langage.

– X est une variable symbolique que l'on appelle variable muette dans la mesure où à la définition de la fonction cette variable n'est pas forcément connue. Elle représente la variable libre de la fonction et peut donc être remplacée par n'importe quelle valeur lors de l'utilisation de la fonction. Elle est alors associée à une variable X0 ou une constante du programme qui remplace symboliquement X par une valeur dans l'expression définissant cette fonction.

– L'expression qui définit la fonction est une expression faisant appel à des opérateurs ou fonctions standards ou déjà définies dans le programme.

– Cette expression doit bien sûr être une fonction de la variable muette ou libre définie ci-dessus.

Remarque. – Sur certains BASIC, il est possible de définir des fonctions travaillant sur des variables chaînes de caractères.

Exemples :

Soit à définir une fonction polynôme du troisième degré du type :

$$Ax^3 + Bx^2 + Cx + D$$

On utilisera la déclaration :

```
10  DEF FNPO(X) = A * X↑3 + B * X↑2 + C * X + D
```

```
 5  INPUT A,B,C,D
10  DEF FNPO(X) = A * X↑3 + B * X↑2 + C * X + D
20  Y = FNPO(1)
30  FOR I = 0 TO 10
40  PRINT FNPO(I) ;
50  NEXT I
60  END
```

Exécution :

```
?      1,2,3,4 ®
□      4 10 26 58 112 194 310 466 668 922 1234
```

– On peut aussi définir des fonctions qui peuvent être obtenues à partir de fonctions standards.

Exemples :

– Si l'on veut obtenir les fonctions trigonométriques en utilisant un angle en degré on définira :

```
10      DEF FNCO(X) = COS(X*π/180)
20      DEF FNSI(X) = SIN(X*π/180)
30      DEF FNTA(X) = TAN(X*π/180)
40      FOR I = 0 TO 60 STEP 15
50      PRINT I ; FNCO(I) ; FNSI(I) ; FNTA(I)
60      NEXT I
70      END
```

Ce programme permet d'imprimer les valeurs de COS, SIN, TAN pour des angles en degrés allant de 0 à 60° par pas de 15°.

Exécution :

		COS	SIN	TAN
	0	1	0	0
Angles	15	0.965925826	0.258819045	0.26949192
en	30	0.866025404	0.5	0.577350269
degrés	45	0.707106782	0.707106781	0.99999999
	60	0.5	0.866025404	1.73205081

– Fonction Logarithme décimal :

D'après ce qui a été vu dans le paragraphe précédent on peut définir :

$$Log_{10}(x) = Log\ x/Log\ 10$$

Le programme suivant imprime les valeurs de Log10 de 1 à 10

```
10      DEF FNLO(X) = LOG(X)/LOG(10)
20      FOR I = 1 TO 10
30      PRINT I ; FNLO(I) ;
40      NEXT I
50      END
```

Exécution :

```
 1      0
 2      0.30102996
 3      0.477121255
 4      0.60259991
 5      0.698970004
 6      0.77815125
 7      0.84509805
 8      0.903089987
 9      0.954242509
10      1
```

AUTRES FONCTIONS MATHEMATIQUES DERIVEES DES FONCTIONS STANDARDS

a) *Fonctions trigonométriques inverses :*

Arc sin (x) : FNAS(X) = ATN(X/SQR (– X * X + 1))
Arc cos (x) : FNAC(X) = – ATN(X/SQR (– X * X + 1)) + π/2
Arc cotg (x) : FNAT(X) = – ATN(X) + π/2

b) *Fonctions hyperboliques*

cos *h* (*x*) : FNCH(X) = (EXP(X) + EXP(– X))/2
sin *h* (*x*) : FNSH(X) = (EXP(X) – EXP(– X))/2
tang *h* (*x*) : FNTH(X) = – EXP(– X)/(EXP(X) + EXP(– X)) * 2 + 1
cotang *h* (*x*) : FNGH(X) = EXP(– X)/(EXP(X) – EXP(– X)) * 2 + 1

c) *Fonctions hyperboliques inverses :*

Arc sin *h* (*x*) : FNHS(X) = LOG(X + SQR(X * X + 1))
Arc cos *h* (*x*) : FNHC(X) = LOG(X + SQR(X * X – 1))
Arc tan *h* (*x*) : FNHT(X) = LOG((1 + X)/(1 – X))/2
Arc cotg *h* (*x*) : FNHG(X) = LOG((X + 1)/(X – 1))/2

Applications : Calcul d'un nombre arrondi à un certain nombre de décimales

Dans les exemples que nous avons pu donner jusqu'à présent, les nombres décimaux étaient imprimés avec huit décimales. Dans la plupart des cas, cela est superflu. Pour arrondir un nombre à n décimales, il suffit de multiplier ce nombre par 10^n, de prendre la partie entière et de diviser le nombre obtenu par 10^n.

Exemple :

Soit le nombre 145.7895678 à arrondir à deux décimales.

Si on le multiplie par $10^2 = 100$ on obtient 14578.95678 ; en prenant la partie entière, on obtient : 14578 ; en divisant par $10^2 = 100$ on obtient 145.78.

Si l'on veut d'autre part tenir compte de la décimale suivante (ici 9) on peut rajouter 0.5 au nombre obtenu après la multiplication par 10^2, soit 14579.45678. Ce qui donnera finalement le nombre arrondi à la décimale supérieure : 145.79. Si la décimale suivante avait été un chiffre entre 0 et 4 on aurait arrondi le dernier chiffre à la décimale inférieure.

Programme. – D'après ce que nous venons de voir, on peut définir une fonction AR(X) telle que :

AR(X) = INT (X * 10↑N + 0.5)/10↑N

Soit le programme suivant :

```
10  DEF FNAR(X) = INT (X * 10↑N + 0.5)/10↑N
20  INPUT "NOMBRE DE DECIMALES" ; N
30  INPUT X
40  PRINT X ; "ARRONDI A" ; N ; "DECIMALES ="
45  PRINT FNAR(X)
50  GO TO 20
60  END
```

Exécution :

```
    NOMBRE DE DECIMALES ? 3 ®
?   45.897895 ®
□   45.897895 ARRONDI A 3 DECIMALES = 45.898
    NOMBRE DE DECIMALES ? 2 ®
    0.123456 ®
□   0.123456 ARRONDI A 2 DECIMALES = 0.12
    NOMBRE DE DECIMALES ?0 ®
?   456.125 ®
□   456.125 ARRONDI A 0 DECIMALE 456
```

Remarque. – Une telle fonction est très utile, en particulier pour des applications de gestion, où l'on veut arrondir les résultats définitifs de calculs sur des francs et ou des centimes.

Trouver un zéro réel d'un polynôme du troisième degré :

On sait qu'un polynôme :

$$y = f(x) = a_n x^n + \ \dots \ + a_2 x^2 + a_1 x + a_0$$

aura un zéro réel compris entre x_1 et x_2 si

$$f(x_1) \times f(x_2) < 0$$

En effet, puisqu'il s'agit d'une fonction continue, cela veut dire que $f(x_1)$ et $f(x_2)$ sont de signes opposés et donc que la fonction passe par zéro entre ces deux points. Pour cela, on fera une première recherche d'un intervalle x_1, x_2, dont on est certain qu'il contient un zéro, puis on fera une recherche à 10^{-2} près d'un zéro dans cet intervalle. Une solution est donnée par le programme suivant :

```
  1  INPUT A, B, C, D
  5  DEF FNAR(X) = (X * 100 + 0.5)/100
 10  DEF FNP(X) = A * X↑3 + B * X↑2 + C * X + D
 20  D1 = 1 : D2 = 0.01
 30  X1 = - 10 : X2 = 10
 40  FOR X = X1 TO X2 STEP D1
 50  IF FNP(X) * FNP (X + D1)< = 0 THEN 100
 60  NEXT X
 70  IF D1 = D2 THEN 200
 80  D1 = D1/10
 90  GO TO 40
100  IF D1 = D2 THEN I = X : GO TO 140
110  FOR I = X TO X + D1 STEP D2
120  IF FNP(I) * FNP (I + D2)< 0 THEN 140
130  NEXT I
140  R = FNAR(I)
150  PRINT "RACINE = " ; R
160  GO TO 1
200  PRINT "PAS DE RACINES" DANS L'INTERVALLE" ;
     X1 ; X2
210  GO TO 1
220  END
```

Exécution :

```
?    3, 7, 1, - 2 ® (3 x³ + 7 x² + x - 2)
     RACINE = - 2.00
?    - 4, 4, 3, 4, ® (4 x³ + 4 x² + 3 x + 4)
     RACINE = 1.75
?    0, 1, 1, - 2 ® (x² + x - 2)
     RACINE = - 2.00
```

Les fonctions de plusieurs instructions

Certaines réalisations de BASIC permettent de définir des fonctions de plusieurs paramètres associés à un calcul qui nécessite plusieurs instructions.

Là aussi la fonction est définie par un nom de fonction associé à une liste de paramètres. La fin de la définition de la fonction est indiquée par une instruction de fin.

Le retour à l'instruction appelante s'effectue comme pour les sous-programmes par une instruction de retour (RETURN)

La structure de programme est alors :

DEF FN nom de fonction (liste de paramètres)
Instructions contenant au moins une instruction
RETOUR paramètre
FIN FN

Une telle structure n'est pas standard et est disponible par exemple sur le BASIC Z 80 de ZILOG sous la forme suivante :

DEF FN identificateur (p1, p2, ...pn)
corps de fonction avec une instruction
RETURN pi
FNEND

Les *p*1....*pn* sont des paramètres transmis à la fonction.

Remarque. – Une telle structure sera très appréciée des utilisateurs connaissant d'autres langages de programmation évolués, car elle comble une des lacunes essentielles du langage BASIC standard en permettant de définir des blocs d'instructions appelables à partir d'un programme et ne transmettant que les paramètres nécessaires au calcul.

Exercices

1. *Définir une fonction qui calcule :*

$$SL(A) = 2.549 \log (A + A^2 + \frac{1}{A})$$

log représente le logarithme décimal.

2. *Utiliser la fonction de 1 pour calculer :*

$$R = X + \log X + 2.549 \log (X + X^2 + \frac{1}{X})$$

Corrigé

1. 10 DEF FNSL(A) = 2.549 * LOG (A + A * A + 1/A) / LOG(10)

2.
```
 1 INPUT X
10 DEF FNLO(X) = LOG (X) / LOG (10)
20 DEF FNSL(A) = 2.549 *  FNLO (A + A * A + 1/A)
30 R = X + FNLO(X) + FNSL(X)
40 PRINT FNLO(X) ; FNSL (X) ; R
50 END
```

Exécution :

```
?  2 ®
   .301029996     2.0711615     4.37314614
?  3 ®
   .4777121255     2.78116412     6.25828537
```

3-3. Les sous-programmes

Il se peut que dans certains programmes le même traitement soit effectué plusieurs fois à différents endroits du programme. La définition de fonction permet de faire référence plusieurs fois au même traitement si celui-ci peut s'exprimer par une expression arithmétique. Cependant, si ce traitement nécessite plusieurs instructions, les fonctions ne sont plus utilisables. Il faut alors avoir recours à la notion de sous-programme ("subroutine" en anglais).

DEFINITION

Un sous-programme est une suite d'instructions réalisant un traitement qui peut être appelé à partir d'un autre programme ou sous-programme, et qui se termine par une instruction de retour au programme appelant.

L'intérêt d'un sous-programme est double :

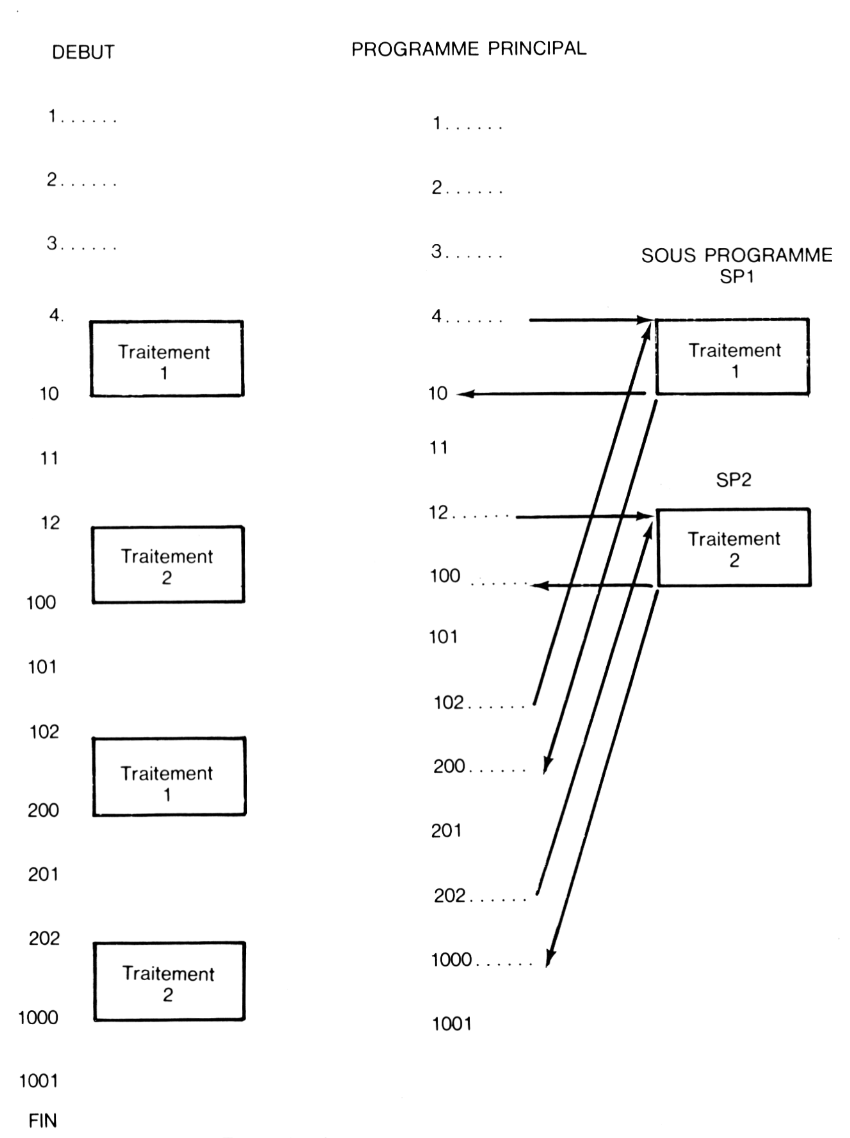
– Il permet d'éviter la duplication des instructions effectuant un traitement donné lorsque ce traitement est nécessaire en plusieurs endroits d'un même programme ;

– Il permet de réutiliser des parties de programmes déjà testés et nécessaires à d'autres programmes. Il permet aussi de construire de nouveaux programmes avec des modules déjà « préfabriqués ». Bien que cela ne soit pas vraiment le cas en BASIC standard, un sous-programme peut être considéré comme une « boîte noire » dans laquelle on entre des données et qui ressort des résultats sans que l'on ait à se préoccuper de ce qui se trouve à l'intérieur du sous-programme.

Suppposons qu'un programme se présente de la façon décrite ci-dessous.

On voit ici les traitements 1 et 2 se répéter deux fois dans ce programme. On peut donc considérer Traitement 1 et Traitement 2 comme des sous-programmes.

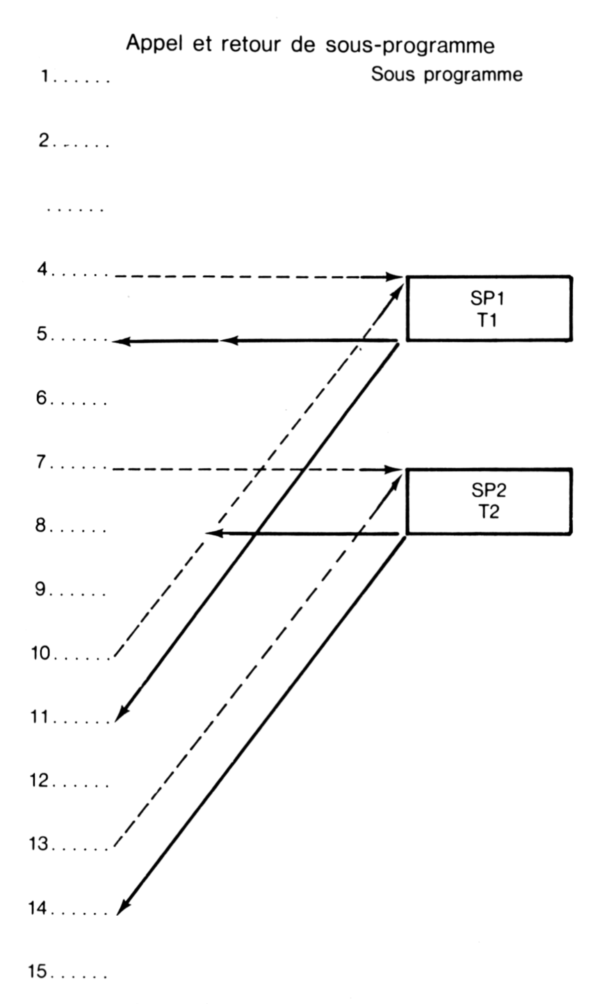
Ainsi le schéma sur la droite représente la nouvelle structure : il n'existe plus qu'une seule version des instructions composant les traitements 1 et 2.



Exemple de structure de sous-programme

A l'instruction 4 il y a un branchement vers le sous-programme SP1, et lorsque ce traitement est terminé le retour s'effectue en 10, qui est l'instruction qui suit immédiatement l'instruction de branchement. Le même procédé est utilisé en 102 avec retour en 200 après exécution du même sous-programme SP1.

En fait les instructions du programme appelant (programme principal) pourraient être numérotées de la façon suivante :



Ainsi peut être énoncée cette *règle :*

Le retour d'un sous-programme s'effectue toujours à l'instruction qui suit l'instruction d'appel ou de branchement au sous-programme.

LES INSTRUCTIONS D'APPEL ET DE RETOUR D'UN SOUS-PROGRAMME

Ces instructions sont très simples.

L'appel ou le branchement à un sous-programme est :

– APPEL SP NN ou ALLER ASP NN

(SP = sous-programme), NN étant le numéro de la première instruction du sous-programme.

En BASIC, cette instruction est :

– GOSUB NN

Le retour d'un sous-programme est tout simplement une instruction :

RETOUR (RETURN).

Dans ce cas il n'y a pas besoin de préciser le numéro d'instruction d'après la règle énoncée ci-dessus.

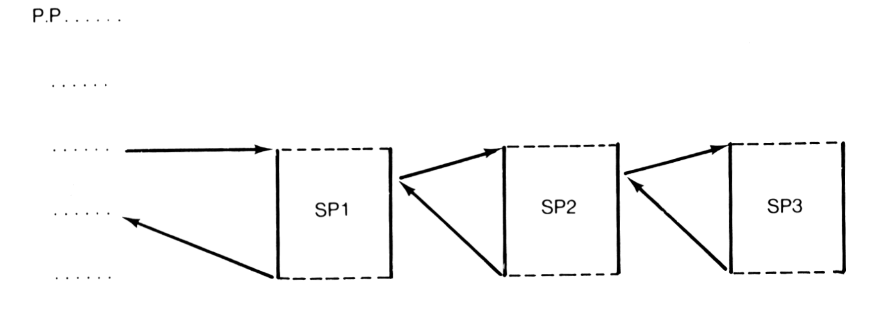
Remarques. – Il peut y avoir plusieurs points d'entrée dans un sous-programme.

– Il peut y avoir plusieurs instructions de retour dans un même sous-programme.

– Bien qu'en BASIC il soit possible de sortir d'un sous-programme par une instruction ALLER A sans passer par une instruction de RETOUR, ceci est une *TRES MAUVAISE* technique de programmation. En principe, on ne doit sortir d'un sous-programme que par une instruction de retour.

EMBOITEMENT DE SOUS-PROGRAMMES

Il est possible d'appeler un autre sous-programme à partir d'un sous-programme. Il peut ainsi y avoir l'emboîtement de plusieurs niveaux de sous-programmes.



Programme principal et structure de sous-programmes

Ici, il existe trois niveaux de sous-programmes.

Le programme principal appele SP1, qui appelle SP2, qui appelle lui-même SP3.

L'on conçoit que dans des structures de ce type il puisse exister une limite dans le nombre de niveaux : en pratique cette limite dépend de l'interpréteur et dans la plupart des systèmes il n'y en a pas.

Exemples d'utilisation de sous-programmes

Soit un programme qui rentre deux paramètres quantitatifs concernant une population d'individus (par exemple le poids et la taille).

Il s'agit d'écrire un programme qui calcule la moyenne, la variance et l'écart type de ces deux paramètres.

On rappelle que si l'on a n individus, x_i représentent la mesure concernant l'individu i, la moyenne est par définition :

$$m = \frac{\sum_{i=1}^{n} x_i}{n}$$

et la variance théorique est :

$$V = \frac{\sum_{i=1}^{n} x_i^2}{n} - m^2$$

son estimation est :

$$V = \frac{\Sigma x^2 - (\Sigma x)^2/n}{n - 1}$$

on sait que l'écart type $E = \sqrt{V}$.

Dans ce cas, il est évident que l'on doit considérer le calcul de la moyenne et de la variance comme un sous-programme qui sera appelé deux fois : une fois pour le premier paramètre et une seconde fois pour le deuxième paramètre.

Nous allons donc commencer par écrire un tel sous-programme.

Nous avons déjà vu un programme de calcul de la moyenne. Pour calculer les variances il suffit de calculer la somme des carrés Σx_i^2.

Si N est le nombre d'éléments on a donc le sous-programme correspondant :

```
200      S = 0
210      V = 0
220      E = 0
230      FOR I = 1 TO N
240      S = S + X(I)
250      V = V + X(I) * X(I)
260      NEXT I
270      M = S/N
280      V = V/N - M↑2 ou V = (V - S↑2/N)/(N - 1)
290      E = SQR(V)
300      RETURN
```

Le programme principal est alors chargé de lire les données et d'imprimer les résultats. Une première solution consiste à lire les données successivement en utilisant le tableau X.

```
 10  INPUT N
 20  DIM X(N)
 30  REM LECTURE DU PREMIER PARAMETRE
 35  PRINT "POIDS"
 40  FOR I = 1 TO N
 50  INPUT X(I)
 60  NEXT I
 70  GOSUB 200
 90  PRINT "MOYENNE" ; M ; "VARIANCE" ; V ; "ECART
     TYPE" ; E
100  REM LECTURE DU 2E PARAMETRE
105  PRINT "TAILLE"
110  FOR I = 1 TO N
120  INPUT X(I)
130  NEXT I
140  GOSUB 200
160  PRINT "MOYENNE" ; M ; "VARIANCE" ; V ; "ECART
     TYPE" ; E
170  END
```

Exécution :

```
?    5 ®      nombre d'individus
POIDS
?    65
?    70
?    80
?    75
?    72
□    MOYENNE 72.4 VARIANCE 25.0399901 ECART TYPE
5.00399742
TAILLE
?    1.60
?    1.75
?    1.80
?    1.70
?    1.65
□    MOYENNE 1.7 VARIANCE 4.99999766 E - 03 ECART
TYPE 0.07071606
```

Deuxième solution

Dans l'exemple précédent, il y avait lecture successive des données dans un tableau. En fait, si l'on ne veut pas conserver les données, on peut se passer de l'utilisation d'un tableau. On obtient alors le programme donné ci-dessous. Dans cet exemple on utilise

deux sous-programmes, un premier pour lire les données et accumuler les sommes partielles Σx *et* Σx^2. Le deuxième sous-programme calcule M, V et E et les imprime.

```
10      INPUT N
20      PRINT "POIDS"
30      GO SUB 100
40      GO SUB 200
50      PRINT "TAILLE"
60      GOSUB 100
70      GOSUB 200
80      END
```

Premier sous-programme

```
100     S = 0 : V = 0 : E = 0
110     FOR I = 1 TO N
120     INPUT X
130     S = S + X
140     V = V + X * X
150     NEXT I
160     RETURN
```

Deuxième sous-programme

```
200     M = S/N
210     V = V/N - M * M ou V = (V - S * S/N)/(N - 1)
220     E = SQR(V)
230     PRINT "MOYENNE" ; M ; "VARIANCE" ; V ;
240     PRINT "ECART TYPE" ; E
250     RETURN
```

Exécution

```
?       5 ®
POIDS
?       60
?       55
?       45
?       65
?       50
□       MOYENNE 55 VARIANCE 49.9999979 ECART
TYPE 7.07106767
TAILLE
?       1.62
?       1.50
?       1.45
?       1.60
?       1.55
□       MOYENNE 1.544 VARIANCE 3.94399777 E-03
ECART TYPE 0.06280126
```

Troisième solution

Supposons que l'on veuille conserver les données d'entrée. Dans ce cas il faut revenir à la première solution et utiliser deux tableaux P et T pour lire les données et un tableau X pour faire les calculs.

```
10    INPUT N
20    DIM P(N), T(N), X(N)
30    FOR I = 1 TO N
40    INPUT P(I), T(I)
50    NEXT I
60    PRINT "POIDS"
70    FOR I = 1 TO N
80    X(I) = P(I)
90    NEXT I
100   GOSUB 200
110   GOSUB 300
120   PRINT "TAILLE"
130   FOR I = 1 TO N
140   X(I) = T(I)
150   NEXT I
160   GOSUB 200
170   GOSUB 300
180   END
190   REM PREMIER SOUS-PROGRAMME
200   S = 0 : V = 0 : E = 0
210   FOR I = 1 TO N
220   S = S + X(I)
230   V = V + X(I) * X(I)
240   NEXT I
250   RETURN
```

Le deuxième sous-programme (300) est identique à celui de la deuxième solution :

```
300   M = S/N
310   V = V/N - M * M ou V = (V - S↑2/N)/(N - 1)
320   E = SQR (V)
330   PRINT "MOYENNE" ; M ; "VARIANCE" ; V ;
340   PRINT "ECART TYPE" ; E
350   RETURN
```

Exécution :

```
?     5 ®
?     56, 1.6 ®
?     60, 1.62 ®
?     65, 1.70 ®
?     70, 1.75 ®
?     72, 1.78 ®
```

□ POIDS
MOYENNE 64.6 VARIANCE 35.83 ECART TYPE 5.98
□ TAILLE
MOYENNE 1.69 VARIANCE 4.959 E-03 ECART TYPE 0.0704

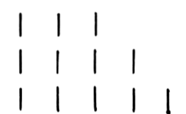
Remarque sur les sous-programmes en BASIC

Les exemples précédents illustrent la difficulté de faire des sous-programmes généraux en BASIC. En effet, comme il n'y a pas de transmission de paramètres entre le programme appelant et un sous-programme, il faut que les variables utilisées dans le sous-programme soient connues du programme principal et inversement. Ceci est un des inconvénients majeurs du langage : il n'est pas possible de développer des sous-programmes paramétrés, ce qui réduit la flexibilité et l'utilité des sous-programmes en BASIC. Il faut cependant remarquer que sur certaines réalisations de BASIC, il est possible de définir des sous-programmes paramétrés.

Nous terminerons ce paragraphe en présentant un programme plus complexe où la décomposition en sous-programmes permet de mieux appréhender, de résoudre et de programmer le problème posé.

Le jeu de Nim

Le jeu de NIM, dit jeu de Marienbad, est joué ici avec douze allumettes disposées sur trois rangées de la manière suivante :



Le jeu consiste à enlever des allumettes suivant les règles énoncées ci-dessous. Le dernier à retirer une ou des allumettes gagne.

L'ensemble constitué par n_1 allumettes dans la première rangée, n_2 dans la deuxième et n_3 dans la troisième peut être représenté par le triplet (n_1,n_2,n_3). La situation de début de jeu est alors représentée par le triplet (3, 4, 5). Le jeu se joue à deux joueurs. Chacun à son tour peut enlever une ou plusieurs allumettes d'une rangée. Celui qui enlève les dernières allumettes gagne (situation [0,0,0]).

Soient deux joueurs X et Y. Une situation (n_1,n_2,n_3) atteinte par X est dite « gagnante » si, quelle que soit l'action de Y, il existe une action de X conduisant à la situation (0,0,0). Lorsqu'un

joueur X a atteint une position gagnante, il peut gagner d'une manière systématique en réagissant à chaque action de son adversaire Y par une action conduisant à nouveau à une situation gagnante. Cette stratégie permet au joueur X d'atteindre éventuellement la situation (0,0,0) et de gagner la partie. La connaissance des situations gagnantes est donc d'une importance cruciale.

Or, on sait déterminer dans le cas du jeu de NIM si une situation est gagnante ou pas comme suit (cet algorithme peut être généralisé à un nombre quelconque d'allumettes, de rangées, et à une disposition initiale quelconque. Essayez) :

1. Soit $b_1b_2b_3$ l'équivalent binaire de n_1, $c_1c_2c_3$ celui de n_2, et $d_1d_2d_3$ celui de n_3.

2. Soit $r_1 = b_1 + c_1 + d_1$
$r_2 = b_2 + c_2 + d_2$
$r_3 = b_3 + c_3 + d_3$

3. Si r_1, r_2 et r_3 sont tous pairs, alors (n_1, n_2, n_3) est une situation gagnante, autrement pas.

Par exemple (3, 4, 5) n'est pas une situation gagnante.

En binaire 3 est $b_1b_2b_3 = 011$
4 est $c_1c_2c_3 = 100$
5 est $d_1d_2d_3 = 101$

et $r_2 = 1 + 0 + 0$ est impair.

Par contre (1, 4, 5) est une situation gagnante.

En binaire 1 est $b_1b_2b_3 = 001$
4 est $c_1c_2c_3 = 100$
5 est $d_1d_2d_3 = 101$

et $r_1 = 0 + 1 + 1 = 2$
$r_2 = 0 + 0 + 0 = 0$ tous pairs.
$r_3 = 1 + 0 + 1 = 2$

– Ecrire un programme qui simule des parties de Nim pour un joueur A jouant contre un joueur B (l'ordinateur ou plutôt le programme). Le joueur A entre son choix sous forme de L (nombre d'allumettes) à retirer sur la ligne M. Un premier programme considère que le joueur B choisit au hasard entre les divers coups possibles. Dans un deuxième programme le joueur B essaie d'atteindre une situation gagnante ; si ce n'est pas possible, le joueur B enlève une allumette dans la première rangée possible. On vérifiera que B gagne chaque fois qu'il commence, lorsqu'il part d'une position gagnante au premier coup.

1. *Programme de jeu de Nim aléatoire*

```
10    INPUT "NOMBRE DE RANGEES" ; N
15    DIM A(N)
20    FOR I = 1 TO N
30    INPUT "NOMBRE D'ALLUMETTES" ; A(I)
40    NEXT I
50    GOSUB 410
110   INPUT "A VOUS" ; L,M
120   IF L > N THEN 110
125   IF A(L) = 0 THEN 110
130   GOSUB 700
135   IF G = 1 THEN 140
136   PRINT "VOUS AVEZ GAGNE"
137   GO TO 20
140   PRINT "A MOI !"
145   L = INT (N * RND(1) + 1)
150   IF A(L) = 0 THEN 145
160   M = INT (A(L) * RND (M) + 1)
165   IF M = 0 THEN 160
170   GOSUB 700
180   IF G = 1 THEN 110
190   PRINT "J'AI GAGNE"
200   GO TO 20
```

Sous-programme de soustraction de M allumettes à la ligne L

```
700   A(L) = A(L) - M
705   IF A(L) < 0 THEN A(L) = 0
710   GOSUB 410
745   G = 0
750   FOR I = 1 TO N
760   IF A(I) = 0 THEN 780
770   G = 1
780   NEXT I
790   RETURN
```

2. *Programme de jeu de NIM avec recherche d'une solution gagnante*

```
10    INPUT "NOMBRE DE RANGEES" ; N
15    DIM A(N), B(N, 10), R(10)
20    FOR I = 1 TO N
30    INPUT "NOMBRE D'ALLUMETTES" ; A(I)
40    NEXT I
50    GOSUB 400
60    FOR I = 1 TO N
```

```
 70   C = A(I)
 80   GOSUB 500
 90   NEXT I
100   GOSUB 600
110   INPUT "A VOUS" ; L,M
120   IF L > N THEN 110
125   IF A(L) = 0 THEN 110
130   GOSUB 700
135   IF G = 1 THEN 140
136   PRINT "VOUS AVEZ GAGNE"
137   GO TO 20
140   PRINT "A MOI !"
145   FOR J = 10 TO 0 STEP - 1
146   IF R(J) = 0 THEN 200
150   IF R(J)/2 = INT (R(J)/2) THEN 200
160   FOR I = 1 TO N
170   IF A(I) = 0 THEN 190
180   IF B(I, J) = 1 THEN 210
190   NEXT I
200   NEXT J
201   REM PAS DE SOLUTION GAGNANTE
202   FOR I = 1 TO N
203   IF A(I)< >0 THEN 205
204   NEXT I
205   L = I : M = 1
206   GO TO 220
210   L = I
215   GOSUB 800
220   GOSUB 700
230   IF G = 1 THEN 110
240   PRINT "J'AI GAGNE"
250   GO TO 20
```

Sous-programme d'impression

```
400   REM IMPRESSION DU JEU
410   PRINT
420   FOR I = 1 TO N
430   IF A(I) = 0 THEN 470
440   FOR J = 1 TO A(I)
450   PRINT "I" ;
460   NEXT J
470   PRINT
480   NEXT I
490   RETURN
```

Sous-programme de conversion décimal binaire

```
495     REM CALCUL DE LA VALEUR BINAIRE DE A(I)
500     FOR J = 10 TO 0 STEP - 1
510     B(I,J) = INT (C/2↑J)
520     C = C - B (I,J) * 2↑J
530     NEXT J
540     RETURN
```

Sous-programme de calcul des sommes binaires

```
590     REM CALCUL DES R(I)
600     FOR J = 10 TO 0 STEP - 1
610     R(J) = 0
620     FOR I = 1 TO N
630     R(J) = R(J) + B(I,J)
640     NEXT I
650     NEXT J
660     RETURN
```

Sous-programme de soustraction de M allumettes d'une rangée L

```
700     A(L) = A(L) - M
705     IF A(L)< 0 THEN A(L) = 0
710     GOSUB 410
720     C = A(L) : I = L
730     GOSUB 500
740     GOSUB 600
745     G = 0
750     FOR I = 1 TO N
760     IF A(I) = 0 THEN 780
770     G = 1
780     NEXT I
790     RETURN
```

Sous-programme de recherche d'une configuration gagnante

```
800     I1 = I : J1 = J
805     FOR K = A(I) - 1 TO 0 STEP - 1
810     C = K : I = I1
820     GOSUB 500
830     GOSUB 600
840     FOR J = J1 TO 0 STEP - 1
850     IF R(J)/2 < > INT (R(J)/2) THEN 890
860     NEXT J
865     REM SOLUTION GAGNANTE
870     M = A(I1) - K
880     RETURN
890     NEXT K
895     REM PAS DE SOLUTION GAGNANTE
```

```
900        C = A(I1) : I = I1
910        GOSUB 500
920        GOSUB 600
930        M = 1
940        RETURN
```

4. Les instructions d'aiguillages multiples SUR ... ALLER (ON ... GOTO)

Les instructions de test (SI... ALORS...) permettent de programmer n'importe quel algorithme et en particulier les aiguillages multiples. Cependant, dans certains cas, il est intéressant de pouvoir remplacer une suite de tests sur la même variable ou expression par une seule instruction permettant de renvoyer le contrôle vers différentes zones du programme traitant une condition particulière. En BASIC étendu, cette possibilité est offerte par l'instruction SUR (ON)... ALLER A (GO TO).

Syntaxe et sémantique de l'instruction

Du point de vue syntaxique, cette instruction se présente sous la forme générale suivante :

SUR Expression arithmétique ALLER A liste d'étiquettes
ON Expression arithmétique GO TO liste d'étiquettes

L'expression arithmétique est évaluée sous forme *entière,* c'est-à-dire que le résultat du calcul est tronqué à sa partie entière. La liste d'étiquettes est une suite de nombres séparés par des virgules et correspondant à des numéros d'instructions correspondant à l'aiguillage.

Si la valeur de l'expression est égale à 1, le contrôle sera donné à l'instruction correspondant à la première étiquette. Si la valeur de l'expression est 2, le contrôle sera donné à la deuxième étiquette... ; si la valeur de l'expression est N, le contrôle sera donné à la N^{e} étiquette.

Enfin, si la valeur de l'expression ne correspond pas à une valeur associée à une étiquette de la liste (valeur $\leqslant 0$ ou supérieure au nombre d'étiquettes) alors, comme dans une instruction de test, le contrôle est donné à l'instruction en séquence.

Remarques. – Sur certains BASIC, il est nécessaire que l'expression donne un résultat entier, sinon l'aiguillage n'a pas lieu et l'instruction en séquence est exécutée. D'autres considèrent une valeur négative comme une erreur à l'exécution.

Exemple :

```
10      ENTRER I,N
20      SUR I ALLER A 40, 60, 80
30      IMPRIMER "ALEATOIRE" ; N ; " = " ; ALE(N) ;
35      ALLER A 90
40      IMPRIMER "EXP" ; N ; " = " ; EXP(N) ;
50      ALLER A 90
60      IMPRIMER "LOG" ; N ; " = " ; LOG (N) ;
70      ALLER A 90
80      IMPRIMER "RAC" ; N ; " = " ; RAC(N) ;
90      FIN
```

Ce programme demande d'entrer deux valeurs : la première (I) sert de variable de contrôle à l'aiguillage, la deuxième est une variable servant de paramètre à une fonction. Si I vaut 1, on imprimera la valeur de e^N ; si I vaut 2, on imprimera la valeur de Log (N) ; si I vaut 3, on imprimera la valeur de $\sqrt{N}$. Si I prend une autre valeur, on imprimera une valeur aléatoire. A titre d'exemple, ce programme a été testé en BASIC sur un micro-ordinateur CBM.

```
 10     INPUT I,N
 20     ON I GO TO 40,60,80
 30     PRINT "ALEATOIRE" ; N ; " = " ; RND(N)
 35     GO TO 10
 40     PRINT "EXP" ; N ; " = " ; EXP(N)
 50     GO TO 10
 60     PRINT "LOG" ; N ; " = " ; LOG (N)
 70     GO TO 10
 80     PRINT "RAC" ; N ; " = " ; SQR(N)
 90     GO TO 10
100     END
```

Les exécutions suivantes ont été obtenues :

```
?       1,2 ®
        EXP 2 = 7.3890561
?       2,2 ®
        LOG 2 = 0.693147181
?       3,2 ®
        RAC 2 = 1.41421356
?       4,5 ®
        ALEATOIRE 5 = 0.652794473
?       1.4,1 ®
        EXP 1 = 2.71828183
?       0.2,1 ®
        ALEATOIRE 1 = 0.461619015
?       2.4,3 ®
```

```
        LOG 3 = 1.09861229
?       2.78,9 ®
        RAC 9 = 3
?       79.45,10 ®
        ALEATOIRE 10 = 0.021016756
?       - 1,2 ®
        ILLEGAL QUANTITY ERROR IN 20
        (ERREUR QUANTITE ILLEGALE EN 20)
```

On en conclut donc que pour cette réalisation de BASIC toutes les règles énoncées sont vraies. La seule contrainte est que les valeurs négatives de l'expression ne sont pas acceptées. Dans ce cas si l'on ne connaît pas à priori la valeur de l'expression, il sera utile de faire un test avant toute instruction SUR (ON) pour s'assurer que l'expression ne prend pas de valeur négative et de prévoir un message d'erreur correspondant.

Autres exemples

– Si dans le programme précédent l'on rajoute les instructions :

```
15      K = I↑2 + 1
20      ON K GO TO 40,60,80
```

A l'exécution on obtient :

```
?       - 2, 2 ®
        ALEATOIRE 2 = .786648867
?       1,2 ®
        LOG 2 = 0.693147181
?       0,1 ®
        EXP 1 = 2.71828183
?       1.5,2 ®
        RAC 2 = 1.41421356
```

Dans ce cas on peut donc rentrer des valeurs négatives sans erreur à l'exécution.

– Si l'on remplace l'instruction 15 par :

```
15      K = SGN(I) + 1
```

SGN est une fonction qui donne le signe de I, qui vaut – 1 si le signe est négatif, + 1 si le signe est positif, 0 si la valeur est 0. On obtient alors les exécutions suivantes :

```
?       - 1,2 ®
        ALEATOIRE 2 = .714959655
?       0,2 ®
        EXP 2 = 7.3890561
?       2,4 ®
        LOG 4 = - 1.38629436
```

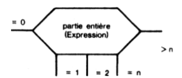
Dans ce cas, on ne peut jamais calculer la fonction racine carrée, car la valeur maximum de K est 1 + 1 = 2.

L'exemple précédent est un aiguillage qui permet de tester trois éventualités : valeur d'une expression < 0, $= 0$ et > 0.

On peut la représenter par un test à trois branches

< 0 Expression > 0
= 0

Dans d'autres langages (FORTRAN en particulier), ce test s'appelle un test arithmétique. En ce qui concerne le cas général d'un aiguillage à n sorties, il n'y a pas de représentation standard, mais on peut par exemple utiliser le schéma suivant :



Les aiguillages vers des sous-programmes

L'instruction SUR (ON), lorsqu'elle existe, est également utilisable pour des aiguillages vers des sous-programmes. Dans ce cas, la syntaxe est identique à celle de l'instruction SUR, mais le mot clé ALLER A (GO TO) est remplacé par APPEL SP (appel sous-programme) ou GOSUB en BASIC.

SUR expression APPEL SP liste d'étiquettes
ON expression GOSUB liste d'étiquettes

L'intérêt de cette instruction est de permettre l'appel de sous-programmes différents suivant la valeur d'une expression ou d'une variable.

Au retour du sous-programme le contrôle est alors donné à l'instruction suivant l'instruction SUR (ON).

Comme, d'autre part, cette instruction est celle qui est exécutée si l'expression ne correspond pas à une des étiquettes, il faut donc s'assurer que le traitement désiré s'effectue bien dans le cas où il y a une expression donnant une valeur ne correspondant pas à l'exécution d'un sous-programme.

La récursivité en BASIC

Certaines réalisations de BASIC permettent de définir des programmes récursifs. C'est le cas notamment des fonctions multilignes sur le BASIC développé par ZILOG. Exprimée de manière simple, la récursivité permet de définir une fonction par une expression, ou une suite d'instructions, qui utilise elle-même cette fonction.

Ainsi, la fonction factorielle peut être définie récursivement par :

$$FAC(N) = FAC(N - 1) \times N$$

De même, pour résoudre le problème de l'inversion d'une chaîne de caractères C$ on peut utiliser la fonction multiligne suivante :

```
DEF FNI$(C$)
IF LEN(C$) < = 1 THEN RETURN C$
RETURN FNI$(C$[2]) + C$[1,1]
```

Cette fonction retourne d'abord le deuxième caractère de la chaîne C$, auquel on ajoute par concaténation le premier caractère de cette chaîne. Si l'on avait C$ = ABCDE, la première itération retourne BCDEA et finalement on obtient à la dernière itération EDCBA.

5. UN EXERCICE DE STYLE : LE PROBLEME DES HUIT REINES

Pour terminer ce chapitre, nous allons étudier un problème un peu plus difficile. Au jeu d'échecs, il est possible de placer huit reines sur un échiquier sans qu'aucune de ces reines ne soit menacée par une autre. On rappelle qu'aux échecs, une reine contrôle toutes les positions se trouvant sur la même ligne, la même colonne et les deux diagonales passant par la position de la reine. Ce problème possède plusieurs solutions et a longtemps fait l'objet de recherches tant de la part des joueurs d'échecs que des mathématiciens. Ainsi, au milieu du XIXᵉ siècle il n'y avait encore que 40 solutions répertoriées.

Le mathématicien Gauss lui-même pensait qu'il y avait 76 solutions. On sait aujourd'hui qu'il existe en fait 92 solutions, mais ce résultat a simplement été obtenu par simple énumération, et il n'y a pas à notre connaissance de démonstration mathématique le prouvant. On se propose donc d'écrire un programme BASIC permettant de trouver ces 92 solutions.

Nous conseillons donc au lecteur de chercher d'abord un algorithme permettant d'obtenir ce résultat. Bien que le programme ne soit pas très long, nous pensons qu'il s'agit d'un problème assez complexe, et il est donc nécessaire d'en présenter une analyse assez détaillée.

Définition d'une position sans conflit

Il n'est bien sûr pas raisonnable, même avec un ordinateur, de tester toutes les configurations possibles de disposition de huit reines sur un échiquier (même en ne mettant qu'une seule reine par colonne il y a plus de 10^9 possibilités pour lesquelles il faudrait rechercher les solutions convenables !).

Il faut donc rechercher une stratégie de placement des reines en tenant compte des reines déjà placées. Il est en effet évident qu'on ne peut pas placer une reine sur la même colonne ou sur la même ligne qu'une reine déjà placée. L'algorithme de placement doit donc supposer que l'on place au plus une reine par colonne. Ensuite, à l'intérieur de chaque colonne, l'algorithme devra donc rechercher la ou les lignes susceptibles de convenir. Enfin, avant de placer une reine sur une ligne, il faudra s'assurer qu'elle n'est pas menacée sur une des diagonales !

On appelle *position sans conflit* une telle position : c'est une position qui, compte tenu des reines déjà placées, permet de placer une nouvelle reine sans qu'elle n'y soit menacée par celles qui ont été déjà placées.

Principe de l'algorithme

Lorsque l'on est arrivé à la huitième colonne et que l'on y trouve une position sans conflit, l'on a donc une solution répondant au problème.

Il s'agit bien sûr de les trouver toutes. Pour cela on va utiliser un algorithme dit de retour en arrière ("backtracking"). Cet algorithme consiste lorsque l'on a trouvé une solution ou lorsque l'on en a pas trouvé pour une colonne donnée, à une étape quelconque de l'algorithme, à revenir à la colonne précédente pour essayer d'y trouver une autre solution sans conflit, et de repartir en avançant jusqu'à la dernière colonne ou jusqu'à une impossibilité. Lorsque cette opération a été répétée jusqu'à la colonne 1, c'est qu'il n'y a plus de solution pour cette position de reine en colonne 1. On change alors la place de la reine en colonne 1 et on répète le processus jusqu'à ce que la reine de la colonne 1 ait été placée sur les huit lignes de cette colonne.

L'algorithme peut alors être résumé aux étapes suivantes :

1. Placer la première reine sur la colonne 1 (C = 1).
2. Si C⩾8 imprimer la solution et aller en 5.
3. Placer une reine sur la ligne 1 de la colonne C (L = 1).
4. Si l'on a une position sans conflit, passer à la colonne suivante (C = C + 1) et aller en 2.
5. Si la reine de la colonne C est sur la ligne 8 (L = 8) passer à la colonne précédente C = C − 1.
6. Si C = 1 et L = 8 terminé.
   Sinon faire L = L + 1.
7. Si L > 8 aller en 5.
   Sinon aller en 4.

Programmation

La programmation diffère un peu de l'algorithme présenté ci-dessus, car l'on a utilisé un tableau CO pour représenter la position d'une reine dans une colonne : ainsi CO(I) représente la i^e colonne et sa valeur représente la ligne sur laquelle se trouve la reine dans cette colonne. (Si CO(I) = 3 la reine de la i^e colonne se trouve sur la troisième ligne.) D'autre part, pour tester l'absence de conflit sur une diagonale, il suffit de vérifier que les reines ne sont pas sur des droites de pente + 1 ou − 1.

Si l'on a deux reines de position $x1$, $y1$ et $x2$, $y2$, il suffit donc de tester que :

$$\left|\frac{y1 - y2}{x1 - x2}\right| \neq 1$$

Soit $|y1 - y2| - |x1 - x2| \neq 0$

Ceci est effectué par l'instruction 110.

Pour l'impression des résultats, une case contenant une reine est mise à 1, sinon elle est à zéro.

Programme permettant d'obtenir toutes les configurations de huit reines sur un échiquier

```
 10    DIM CO(8), E(8)
 20    NS = 1
 40    FOR I = 1 TO 8
 50    CO(1) = I
 60    C = 1
 65    L1 = 1
 70    FOR L = L1 TO 8
 75    REM VERIFIER QU'IL N'Y A PAS DE CONFLIT
 80    FOR K = 1 TO C
 90    DL = ABS (CO(K) - L)
100    IF DL = 0 THEN 140
110    IF DL = ABS (C + 1 - K) THEN 140
120    NEXT K
130    CO(C + 1) = L : GO TO 200
140    NEXT L
145    REM RECUL D'UNE COLONNE
150    C = C - 1
160    IF C = 0 THEN 300
170    L1 = CO (C + 1) + 1
180    IF L1 > 8 THEN 150
190    GO TO 70
195    REM AVANCER D'UNE COLONNE
200    C = C + 1
205    IF C< = 7 THEN 65
```

```
206        PRINT " SOLUTION " ; NS
207        REM IMPRESSION D'UNE SOLUTION
210        FOR J = 1 TO 8
220        FOR K = 1 TO 8
230        E(K) = 0
240        E(CO(J)) = 1
250        PRINT E(K);
260        NEXT K
270        PRINT
280        NEXT J
290        NS = NS + 1
295        GO TO 150
300        NEXT I
```

Exécution :

Nous donnons ici simplement les premières solutions ; si le programme est correct, il doit en imprimer 92.

Solution 1

```
1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
```

Solution 2

```
1 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0
```

Solution 3

```
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0
```

Solution 4

```
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
```

Solution 5

```
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
```

Solution 6

```
0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 0 0 1 0 0 0 0
```

La dernière solution est
Solution 92

0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0
0	0	0	0	1	0	0	0

CONCLUSION

Dans ce chapitre nous avons présenté les dernières caractéristiques essentielles du langage BASIC. L'utilisateur doit maintenant être capable d'envisager la programmation de n'importe quel problème qui peut être résolu par des méthodes algorithmiques.

Nous avons vu en particulier comment traiter des données structurées en listes ou en tableaux, ainsi que les possibilités d'utiliser et de définir de nouvelles fonctions mathématiques.

Enfin, nous avons présenté et étudié la notion de sous-programme. Toutes ces notions sont d'ailleurs communes à tous les langages évolués. Il faut noter cependant une certaine faiblesse du langage BASIC standard pour ce qui est de la définition de sous-programmes paramétrés. Cette carence est quelquefois contournée sur certains dérivés de BASIC en introduisant les notions de variables globales ou locales que nous n'avons pas développées ici.

Les notions que nous développerons dans le chapitre suivant sont plus spécifiques et ne sont pas en général complètement standardisées.

Nous proposons à la fin de ce chapitre un certain nombre d'exercices ou de problèmes non corrigés.

Exercices non corrigés

1. *Le système de numération romain utilise les symboles M, D, C, L, X, V, I pour représenter les nombres 1 000, 500, 100, 50, 10, 5, 1.*
En « vieux romain » un nombre se représentait par une séquence de ces symboles écrits de la gauche vers la droite et la valeur du nombre était la somme des valeurs associées à chaque symbole.
En « romain moderne », les symboles C, X, I, peuvent précéder un symbole de plus grande valeur et, dans ce cas, le symbole C, X, I, doit être retranché de la valeur correspondante.

Exemples :

Décimal	Vieux romain	Romain moderne
4	IIII	IV
9	VIIII	IX
91	LXXXXI	XCI

a) *Ecrire un programme qui lit un nombre en vieux romain et imprimer la valeur décimale correspondante. Faire l'opération inverse (décimale – vieux romain).*
b) *Exécuter une addition en vieux romain sans passer par la valeur décimale et imprimer le résultat en vieux romain.*
c) *Faire la même opération en passant par la valeur décimale et imprimer le résultat en vieux romain.*

2. *Ecrire un programme qui réalise les opérations suivantes :*
a) *Lecture en romain moderne, conversion et impression en décimal.*
b) *Lecture d'un nombre décimal, impression en romain moderne.*
c) *Réaliser une addition, une soustraction et une multiplication de deux nombres en romain moderne et imprimer le résultat en romain moderne.*

3. *Ecrire un programme qui lit un nombre en décimal et l'imprimer en toutes lettres.*
Exemple : *Soit 1 291 = « Mille deux cent quatre-vingt-onze ». On testera le programme avec des nombres allant jusqu'à sept chiffres.*

4. *Ecrire un programme qui lit un nombre en clair et imprime le résultat en décimal (opération inverse de la précédente).*

5. *Ecrire un programme qui imprime un calendrier en tenant compte des années bissextiles et du fait que les années de début de siècle (1800, 1900, 2000...) ne le sont pas. On demande également l'impression des jours en clair. On prendra comme exemple les années 1978, 1980 et 2000.*

6. a) *Ecrire un programme qui calcule* x^2 *pour :*

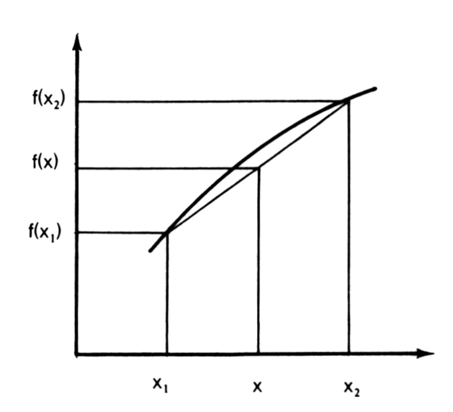
$$x = 1,\ 1.1,\ 1.2,\ 1.3,\ ...,\ 9.8,\ 9.9,\ 10$$

les valeurs x *seront rangées dans un* tableau X *et les valeurs* x^2 *dans un* tableau X2.

b) *En utilisant une méthode d'interpolation linéaire, écrire la suite du programme permettant de calculer des valeurs approchées de* $\sqrt{1}$, $\sqrt{2}$... $\sqrt{100}$.

On rappelle le principe de l'interpolation linéaire :

soit $x_1 < x < x_2$



connaissant les points : $\begin{cases} x_1 \\ f(x_1) \end{cases}$ $\begin{cases} x_2 \\ f(x_2) \end{cases}$

la valeur f(x) *est obtenue en prenant la valeur approchée qui correspond à la droite joignant ces deux points soit :*

$$\frac{f(x_2) - f(x_1)}{f(x) - f(x_1)} = \frac{x_2 - x_1}{x - x_1}$$

7. *Ecrire le programme permettant de calculer la racine carrée négative de 0.25, en démarrant à* $x_0 = -0.6$, *et en appliquant la méthode suivante :*
$F(x) = x^2 - 0.25$
$x_n = x^2_{n-1} + x_{n-1} - 0.25$
La précision relative demandée est de 10^{-3} *au minimum.*

8. *On donne une équation de la forme :*
$F(x) = a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = 0$
Ceci représente un polynôme de degré 6 (n'importe lequel des coefficients peut être nul).
a_0, a_1... a_6 *sont des données.*
Ecrire un programme qui lit ces données, calcule $F(x_0)$, *et imprime les coefficients,* x_0 *et* $F(x_0)$ *pour* x_0 *variant de* -10 *à* $+10$ *par pas de 0.1*

9. *Considérez la séquence suivante :*
1 – E prend la valeur 1
2 – D prend la valeur 5
3 – Remplacer E par 1 + (E.X/D)
4 – Si D = 1 stop. Sinon, soustraire 1 de D et retourner en 3.
Quand l'algorithme s'achève,
$$E = 1 + x + \frac{x^2}{2} + \frac{x^3}{2.3} + \frac{x^4}{2.3.4} + \frac{x^5}{2.3.4.5.} = e^x$$
Ecrire le programme en supposant que X est lu comme une donnée.

10. *Appliquer cette méthode à*
$$\text{cotg}\,(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9}$$
Ecrire le programme.

11. *Comment pourrait-on calculer* 2^{200} *de manière exacte (nombre très grand).*

12. *Le plus petit commum multiple est le plus petit entier qui est divisible par A et B.*
Ecrire le programme permettant de calculer le plus petit commun multiple de deux entiers A et B.

13. *A est un tableau (20,20). Ecrire un programme qui calcule la somme des valeurs absolues des éléments de la* K^e *rangée de A, sauf A (K,K).*
$$\text{SOM ABS} = \sum_{j \neq k} |A\,kj|$$

14. *On a un échantillon de N valeurs et on veut en sélectionner certaines.*
1° On veut sélectionner les valeurs comprises entre m − 2σ *et* m + 2σ *sachant que* m *représente la moyenne des N valeurs et* σ *représente l'écart-type calculé ainsi :*

$$\sigma = \sqrt{\sum_{i=1}^{N} (x_i - m)^2 \, P(X = x_i)}$$

si x_i *représente un élément du tableau on supposera ici que :*

$$P(X = x_i) = \frac{1}{N}.$$

2° *Les valeurs sélectionnées au 1° sont mises dans un deuxième tableau. On veut maintenant sélectionner les* x_i *du deuxième tableau telles que :*

$$m - \frac{m_{i-1}}{2} \leqslant x_i \leqslant m + \frac{m_{i-1}}{2}$$

avec :

$$m_{i-1} = \frac{1}{i-1} \sum_{j=1}^{i-1} x_j$$

3° Ecrire l'organigramme et le programme BASIC qui feront les opérations suivantes :
a) *Lire un tableau A de 50 positions. Imprimer A.*
b) *Faire la première sélection sur le tableau A et transférer les valeurs qui répondent à la sélection dans un tableau B − Imprimer B.*
c) *Faire la deuxième sélection à partir du tableau B, les valeurs résultantes seront transférées dans un tableau C. Imprimer C.*
d) *Trier les valeurs de C et imprimer le tableau trié.*

CHAPITRE V

LES TRAITEMENTS DE FICHIERS EN BASIC

Dans les chapitres précédents, nous avons vu les instructions standards et leurs variantes communes à la plupart des BASIC. Dans ce chapitre, nous allons voir des traitements qui sont d'une certaine manière plus élaborés, mais qui peuvent varier notablement suivant que l'on dispose d'unité de mémoire secondaire de type cassettes magnétiques, de type disques souples ou même sur les plus gros systèmes de disques magnétiques durs. De même, pour les traitements graphiques, cela dépend de l'unité de visualisation dont on dispose : purement alphabétique, semi-graphique ou graphique. Dans certains cas, il est également possible de disposer de traceur de courbe ou d'imprimante permettant d'imprimer du semi-graphique.

Dans le cadre de ce livre, il n'est donc pas possible de voir toutes les variantes ; nous nous limiterons donc à montrer le type de traitement et ce qu'il est possible de réaliser en matière de traitement de fichiers et de traitements graphiques compte tenu des types de systèmes dont on dispose.

ELEMENTS DE BASE SUR LES FICHIERS INFORMATIQUES

1. LA NOTION DE FICHIER

En informatique, la notion de fichier est une notion générale et très simple quant à son contenu.

Définition

Un fichier est une suite d'informations binaires (bits) enregistrées sur un support de mémoire et défini par un identificateur.

De façon plus pratique, on peut dire qu'il s'agit d'une suite de données numériques ou non numériques enregistrées sur un support de mémoire autre que la mémoire centrale. En particulier, dans ce cas, on peut parler de fichiers de cartes perforées, de rubans perforés, de fichiers imprimante (le papier étant lui aussi un support de mémoire), mais, le plus souvent, le support utilisé est de type magnétique. On peut également parler de fichiers « programmes », car, dans la définition donnée ci-dessus, un programme peut être considéré comme un fichier lorsqu'il est stocké sur une mémoire secondaire.

2. LES SUPPORTS DE FICHIERS

On vient de voir que la notion de support de mémorisation est essentielle pour parler de fichier au sens informatique.

Aux débuts de l'informatique, les supports principaux pour les fichiers étaient les cartes perforées et/ou le ruban perforé. Ce type de support, bien qu'encore utilisé, tend à disparaître, car de tels fichiers sont difficiles à manipuler, à stocker et très longs à traiter à cause de la lenteur des lecteurs de cartes ou de rubans.

En sortie nous avons vu que les listages d'imprimantes peuvent être considérés comme des fichiers et, s'il s'agit là d'un type de support qui est bien sûr nécessaire, il s'agit simplement de fichiers qui peuvent être écrits par la machine.

Dans la suite de ce chapitre, nous nous intéresserons donc principalement aux fichiers sur supports magnétiques, qui ont l'avantage de pouvoir être lus et écrits par la machine à des vitesses relativement grandes.

Ils sont d'autre part moins encombrants et d'une manipulation plus facile pour des traitements informatiques. Leur seul défaut est qu'il faut bien sûr avoir un ordinateur pour les traiter aussi bien en lecture qu'en écriture.

Parmi ces supports, on distingue : le support du type *séquentiel :* bandes magnétiques ou cassettes magnétiques et les supports de type *aléatoire* tels que les disques magnétiques.

2-1. Les supports de type séquentiels

Sur les gros et moyens systèmes (y compris les mini-ordinateurs) l'on dispose de *bandes magnétiques* informatiques où l'enregistre-

ment est digital (binaire) et les codages sont normalisés suivant des standards précis. Ce type de bande magnétique permet d'enregistrer jusqu'à plusieurs millions de caractères avec des densités d'enregistrement variant actuellement de 800 à 9 600 BPI (bit per inch). Ce type de support est donc intéressant pour stocker de gros fichiers, en particulier pour les fichiers archives, mais il n'est pas disponible sur des systèmes micro-ordinateurs, car son coût est prohibitif.

Par contre, il existe des supports séquentiels de type *cassettes magnétiques.*

Il faut souligner qu'il existe cependant deux types de systèmes de cassettes : les cassettes digitales et les cassettes audio.

Sur les cassettes digitales, l'information est enregistrée suivant le même principe que sur les bandes magnétiques standard.

De ce fait, elles présentent l'intérêt d'être aussi fiables que des bandes magnétiques, elles peuvent être lues assez rapidement, mais les enregistreurs de telles cassettes sont relativement coûteux. Leur utilisation a considérablement diminué avec l'apparition des disques souples et l'utilisation de systèmes de cassettes audio. Le type de support le plus économique et le plus répandu sur les petits systèmes micro-ordinateurs est actuellement *les cassettes audio,* utilisées avec des enregistreurs de cassettes du commerce. Il faut noter cependant qu'il n'existe aucun standard au niveau du codage des informations et que, suivant les systèmes utilisés, la fiabilité est plus ou moins grande. D'autre part, la lecture de données sur de tels systèmes est un processus assez long, compte tenu des vitesses réduites de lecture et d'écriture.

Pour de petits fichiers traités de façon séquentielle, et notamment pour le stockage de fichier programme, cette solution est tout à fait raisonable.

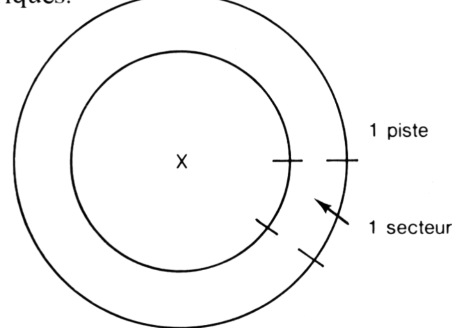
Dans la suite de ce chapitre, nous donnerons des exemples de traitements de fichiers de ce type.

2-2. Les supports de type aléatoire

Par opposition à un support séquentiel, un support est dit aléatoire si l'on peut accéder à l'information contenue sur ce support de manière directe ou aléatoire sans avoir à faire défiler l'ensemble du support du début à la fin pour lire ou écrire une donnée. Cela ne veut pas dire bien sûr qu'un tel support se comporte de façon aléatoire au sens probabiliste : il est toujours commandé par la machine de façon déterministe, mais ce qui est aléatoire, c'est le temps au bout duquel une information aura été lue

ou écrite compte tenu des lectures ou écritures effectuées précédemment.

Parmi ce type de support, on distingue : les disques durs et les disques souples. Dans tous les cas, ces disques sont constitués de pistes concentriques.



Une piste est divisée en secteurs. Chaque piste et chaque secteur sont repérés par un numéro d'ordre que l'on appelle adresse. C'est ce numéro qui permet de spécifier la lecture ou l'écriture de telle piste ou tel secteur.

a) *Les disques durs*

Les disques de ce type sont caractérisés par un support rigide recouvert d'une substance magnétique et enfermé dans une cartouche ou un coffret.

Dans ce cas, les têtes de lecture et d'écriture ne reposent pas sur le support magnétique, ce qui permet d'obtenir des vitesses et des débits plus importants sans provoquer d'usure du support.

Dans cette catégorie on distingue les *disques à têtes fixes,* où il y a autant de têtes de lecture que de pistes. C'est le support qui tourne et les têtes sont fixes. Ceci permet d'accéder plus rapidement à n'importe quel secteur. C'est ce que l'on appelle le *temps d'accès.* Pour ce type de disques, il est de l'ordre de 10 millisecondes en moyenne. Cette catégorie de disques est la plus coûteuse et présente l'inconvénient de ne pas être amovible. Ainsi ils ne peuvent être utilisés pour des fichiers d'archives. Ils ne sont utilisés que sur des systèmes où le temps d'accès est un paramètre crucial pour le fonctionnement du système.

La deuxième catégorie est la catégorie des *disques à têtes mobiles,* où il y a une seule tête de lecture qui peut se déplacer pour se positionner sur la piste désirée. Ceci implique un mouvement du bras qui rallonge le temps d'accès (il peut alors être de l'ordre de 20 à 100 ms, soit 60 ms en moyenne). Ces disques présentent l'avantage d'être moins coûteux et amovibles, ce qui permet à plusieurs supports d'être « montés » à la suite les uns des autres et

de permettre un archivage. Leur capacité varie de plusieurs millions de caractères à plusieurs dizaines de millions de caractères lorsque plusieurs supports sont empilés les uns sur les autres (« dispacks »).

Dans tous les cas, cependant, il s'agit d'unités d'enregistrement relativement coûteuses, bien que les développements de la technologie permettent d'envisager des réductions de prix importantes (disques type « Winchester ») pour des disques non amovibles.

Ce type de disque est nécessaire lorsque l'on veut pouvoir traiter de gros volumes de données et notamment pour les systèmes de bases de données. Sur les gros et moyens systèmes, y compris les systèmes à mini-ordinateurs, ils constituent ce que l'on appelle la « mémoire de masse », qui est partagée entre plusieurs applications et plusieurs utilisateurs.

b) *Les disques souples*

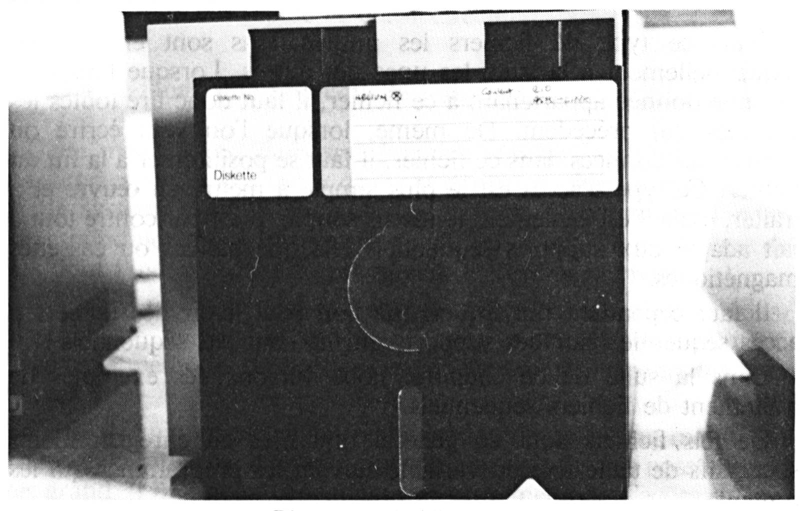
Le principe physique d'enregistrement et d'accès aux informations est le même que pour les disques durs.

Cependant, ce qui diffère, c'est la matière du support (plastique recouvert de substrat magnétique) et la mécanique beaucoup plus simple des enregistreurs.

Il s'agit aussi de disques à têtes mobiles, mais cette fois la tête de lecture repose sur le support.

L'intérêt de ces disques est donc leur faible coût et leur encombrement réduit.

Il existe également différents modèles dont le seul standard actuel soit le standard IBM pour les disques d'une capacité d'environ 250 000 caractères.



Disque souple (disquette)

En dessous, on trouve les « minidisques » qui contiennent environ 100 000 caractères.

Depuis un an, on voit égalemement apparaître les disques « double face double densité » qui permettent de quadrupler les capacités des modèles existants. Sur un disque on peut donc stocker plus d'un million de caractères. Sur un minidisque on peut ainsi aller jusqu'à 500 000 caractères.

Il faut noter cependant que pour les minidisques, il n'existe aucun standard, ce qui peut poser des problèmes de lecture de ces disques par différents systèmes.

Le développement de ces disques permet là aussi d'avoir des petites mémoires de masse associées à un micro-ordinateur.

En particulier, elle permet d'envisager des applications de type gestion sur des systèmes micro-ordinateurs.

La fiabilité de ces supports est actuellement très bonne, et si l'on exclut les dégradations dues à l'usure, à très long terme ils constituent sans aucun doute le système le plus économique pour stocker des fichiers accessibles de façon directe ou aléatoire.

3. LES METHODES D'ACCES AUX FICHIERS

Par méthode d'accès on entend la façon dont un fichier peut être traité par un programme, indépendamment du support sur lequel il se trouve.

3-1. Les fichiers à accès séquentiels

Dans ce type de fichiers les informations sont enregistrées séquentiellement à la suite les unes des autres. Lorsque l'on veut lire une donnée appartenant à ce fichier, il faut donc lire toutes les données qui précèdent. De même, lorsque l'on veut écrire ou ajouter des données dans ce fichier, il faut se positionner à la fin du fichier. Ce type d'accès est le plus simple à mettre en œuvre et à traiter, mais il est également le moins souple. Il est par contre tout à fait adapté aux supports séquentiels tels que bandes ou cassettes magnétiques.

Il faut cependant remarquer que l'on peut avoir des fichiers à accès séquentiels sur des supports qui ne sont pas séquentiels !

Dans la suite de ce chapitre, nous verrons des exemples de traitement de fichiers séquentiels.

De tels fichiers sont en général organisés en enregistrements successifs de taille fixe ou variable suivant les programmes qui les traitent.

3-2. Les fichiers à accès direct

Dans ce type de fichier il suffit de préciser le numéro ou une clé permettant d'identifier la donnée ou l'enregistrement dans lequel se trouve cette donnée pour que l'on y ait accès directement, indépendamment de sa position physique dans le fichier considéré. Cette technique ne peut bien sûr être réalisée de façon pratique que sur des supports de type aléatoire. Dans certains cas, cette méthode d'accès s'appelle d'ailleurs aléatoire, car l'on passe par l'intermédiaire d'une fonction de répartition aléatoire des informations sur le disque, fonction qui a pour paramètre la clé associée à ces données. Ce type d'accès est très souhaitable lorsque l'on veut un accès rapide aux données du fichier.

3-3. Les fichiers à accès indexé

Il s'agit d'une méthode d'accès direct ou « semi-direct » d'un type particulier, puisque l'on doit d'abord rechercher séquentiellement la valeur de la clé dans une table appelée index, et, lorsque l'on a trouvé cette clé, on obtient l'emplacement exact des données ou de l'enregistrement associé.

Ce type d'accès présente l'avantage de donner un accès rapide si les index ne sont pas trop grands, tout en conservant la possibilité de traiter le fichier séquentiellement en balayant les index suivant un ordre défini.

Ce type d'accès sera donc utilisé lorsque l'on veut à la fois traiter des fichiers de manière séquentielle tout en ayant la possibilité d'y accéder de manière rapide et quasi directe.

3-4. La méthode dite séquentielle indexée

Cette méthode est dans son principe similaire à la précédente, mais correspond à des index qui sont définis et gérés complètement par le système d'exploitation de la machine. Dans ce cas, l'index correspond non pas à une valeur de clé isolée, mais à un groupe de clés dont l'index précise le premier et le dernier élément.

Il y a ensuite recherche de l'enregistrement désiré dans un sous-ensemble de fichiers, et cette recherche se fait séquentiellement. Avec cette méthode, on peut envisager également plusieurs niveaux d'index.

Cette technique est donc relativement complexe à mettre en œuvre et ne présente d'intérêt que si le nombre de clés possibles est très grand. Il faut noter cependant que cette méthode d'accès est très

utilisée en gestion, et que l'on commence à la trouver sur des systèmes de type micro-ordinateur orientés vers des applications de gestion.

3-5. L'organisation des fichiers

Quelle que soit la méthode d'accès, les fichiers sont en général organisés en blocs ou entités que l'on appelle des « enregistrements logiques ». Ceci veut dire que la taille du fichier est indépendante de la structure physique des enregistrements sur le support considéré.

L'enregistrement logique est le seul qui soit intéressant du point de vue du programmeur. La taille de ces enregistrements peut être fixe ou variable suivant les applications. La disposition de ceux-ci sur le support peut également être bloquée ou non. Dans le mode bloqué, l'on associe par exemple un secteur à plusieurs enregistrements ce qui permet de mieux utiliser la place disponible.

Lorsqu'il peut être traité séquentiellement, un fichier se termine toujours par un enregistrement particulier que l'on appelle la « fin du fichier » (« end of file »). La détection de cette fin de fichier permet donc de savoir que le traitement est terminé.

4. LES SYSTEMES D'EXPLOITATION DES DISQUES

C'est ce que l'on appelle dans le jargon informatique le DOS (« disc operating system »). Il est constitué par un certain nombre de programmes qui ont été développés par le constructeur et fait partie du logiciel système fourni à l'utilisateur pour la gestion des fichiers sur disques. Un tel système comprendra donc au moins une gestion des tables d'identité des fichiers et de l'espace disponible sur le disque.

Sur les gros et moyens systèmes le DOS peut être très complexe et permettre la gestion de fichiers séquentiels indexés, directs ou séquentiels indexés.

Sur les systèmes micro-ordinateurs, les possibilités sont plus réduites, mais elles évoluent rapidement, notamment sur les systèmes destinés aux applications de gestion.

Sans aller jusqu'à décrire les différents types de DOS que l'on peut rencontrer, il est important de savoir que, pour développer une application de traitement de fichiers, il faut disposer d'un minimum d'opérations que l'on appelle les primitives du système de fichiers et qui sont disponibles pour le programmeur.

Les primitives les plus courantes sont : la création d'un fichier, son ouverture, la lecture et l'écriture d'un enregistrement ou de

données élémentaires, la fermeture d'un fichier, la destruction d'un fichier.

Le DOS comprend en général un certain nombre de programmes utilitaires tels que : la copie de fichiers, la vérification des tables des fichiers, la vérification, le listage des noms de fichiers écrits sur un support, la visualisation du contenu d'un fichier..., etc. Certaines de ces opérations peuvent bien sûr être programmées par l'utilisateur s'il dispose des fonctions élémentaires d'un système de gestion de fichiers.

En ce qui concerne le traitement de fichiers en BASIC, il est bien sûr nécessaire de disposer de ces primitives, qui se traduiront par des instructions particulières.

La deuxième partie de ce chapitre concerne l'étude de ces instructions et leur programmation en utilisant des exemples simples.

5. LE TRAITEMENT DES FICHIERS SUR SUPPORT SEQUENTIEL EN BASIC

Les primitives de traitement de fichiers séquentiels

On suppose par exemple que l'on dispose d'un support séquentiel de type cassette magnétique.

Un conseil pour les utilisateurs qui veulent tester des programmes comportant l'écriture de données dans des fichiers : utiliser toujours des supports vierges pour éviter l'écrasement malencontreux de fichiers ou programmes. D'autre part, avant de se lancer dans le stockage d'un fichier réel, il faut s'assurer que le programme est bien au point.

Moyennant ces quelques recommandations, il faut d'abord présenter les instructions de base nécessaires à la programmation d'instructions de traitement de fichiers séquentiels.

a) *Ouverture d'un fichier*

Cette opération sert à préparer le fichier et à définir un certain nombre de paramètres permettant de savoir s'il s'agit d'une lecture ou d'une écriture et de préciser sur quelle unité doit se trouver le fichier lorsqu'il existe plusieurs enregistreurs. Elle permet également de préciser le nom du fichier.

La syntaxe de cette instruction peut varier suivant les machines, et nous utiliserons l'exemple d'un micro-ordinateur « PET » de Commodore.

Dans ce cas on a une instruction du type suivant :

OUVRIR n° logique de fichier, n° de l'unité, code de lecture écriture, nom de fichier.

– *Le numéro logique* de fichier est un numéro particulier au programme qui servira à référencer ce fichier dans tout le programme. Cela peut être un numéro de 1 à 255, et l'on peut ouvrir simultanément jusqu'à dix fichiers.

– *Le numéro de l'unité* indique le numéro physique de l'enregistrement utilisé si l'on en dispose de plusieurs. Si ce numéro n'est pas indiqué, la valeur par défaut est 1 et correspond à l'enregistreur incorporé à la machine. Ce numéro peut théoriquement aller de 1 à 255, mais bien sûr, si l'on adresse une unité non existante, cela provoque une erreur.

– *Le code de lecture écriture :* si l'on précise 0, c'est une lecture ; 1, une écriture, et 2, une écriture avec écriture d'un marqueur « fin de bande » lors de la fermeture du fichier.

– *Le nom du fichier :* c'est une suite de caractères permettant d'identifier le fichier. Si l'on ne met rien, le nom sera considéré comme « blanc ».

La longueur de ce nom peut aller jusqu'à 80 caractères.

Exemples :

- OPEN 1, 1, 1, "DOSSIER"
  permet d'ouvrir le fichier n° 1 du programme sur l'enregistreur de cassette n° 1 en écriture, sur un fichier appelé « DOSSIER ».
- OPEN 1, 1, 0, "DOSSIER"
  permet d'ouvrir le même fichier en lecture.

b) *Fermeture d'un fichier*

Cette instruction permet d'indiquer que les opérations de lecture ou d'écriture sont terminées sur un fichier. Dans le cas où le fichier avait été ouvert en écriture avec l'option 2, une marque « fin de bande » est écrite.

En principe, tous les fichiers ouverts dans un programme doivent être fermés avant la fin du programme, sinon l'on risque d'obtenir des erreurs à la relecture de ces fichiers. L'instruction ne précise qu'un seul paramètre, c'est le numéro logique du fichier.

FERMER n° logique du fichier
(CLOSE)

Exemple : CLOSE 1
fermer le fichier n° 1.

c) *Ecriture de données dans un fichier :*

Une fois le fichier ouvert en écriture, il suffit de donner un ordre de sortie sur le fichier correspondant en précisant la liste des variables contenant les données à écrire.

Le principe est donc similaire à celui d'un ordre IMPRIMER (PRINT) et l'instruction est en effet identique avec en plus la nécessité de spécifier le numéro du fichier logique concerné.

Ainsi la syntaxe de cette instruction est :

IMPRIMER # n° logique, liste de variables
(PRINT)

Lorsque les noms de variables de la liste sont séparés par des virgules, les données seront enregistrées dans l'ordre précisé et suivies chacune du caractère Retour à la ligne ®, dont le code est 13 en ASCII.

Lorsque les noms de variables sont séparés par des points virgules, les valeurs de ces variables sont enregistrées sans séparateur ®. Cependant, si l'on veut pouvoir relire des données avec un ordre ENTRER, il faut insérer un séparateur ® après le 79e caractère (voir ci-après).

d) *La lecture des données d'un fichier :*

Elle peut se faire suivant deux techniques que l'on a déjà vues pour l'entrée au clavier. La première technique est celle qui concerne la lecture d'une donnée dans son ensemble, qu'elle soit numérique ou non. Pour cela, on utilise l'ordre ENTRER (INPUT).

La structure de cette instruction est alors :

ENTRER # n° fichier, liste de variables
(INPUT)

Ceci permet de lire des données comme si elles venaient du clavier, mais à partir d'un fichier. Les variables peuvent être numériques ou chaînes de caractères. Cela implique en particulier qu'il n'y ait pas de séquences de caractères de longueur supérieure à 79 entre deux caractères ® (retour à la ligne).

Cette contrainte est particulière à la machine utilisée et pourrait être facilement supprimée sur d'autres versions.

L'autre façon de lire des données est de lire caractère par caractère, c'est donc l'instruction OBTENIR (GET) qui est utilisée.

Il en existe deux formes suivant que l'on veut lire du numérique ou des caractères.

La première forme est la suivante.

GET # n° fichier logique, variable numérique.

Dans ce cas, si l'on a un caractère de 0 à 9, la variable contient la valeur correspondante. S'il s'agit d'un caractère + − ou blanc, la variable contient la valeur 0 ou − 1.

Tout autre caractère donnera une erreur (y compris la fin de fichier).

La deuxième forme est la plus générale et utilise une variable chaîne de caractères.

GET # n° de fichier logique, variable chaîne

Dans ce cas, la variable contient le caractère lu, et tous les caractères sont acceptés.

Cette instruction permet en particulier de lire des données qui font plus de 79 caractères sans séparateurs ®.

Le seul problème est de savoir quand s'arrêter. Pour cela, il faut pouvoir tester la fin d'un fichier ou la fin de la bande. De même, il faut pouvoir tester s'il y a eu des erreurs de lecture.

Ceci est effectué en général pour le test d'un code d'état.

e) *Utilisation d'un mot d'état*

Le mot d'état est un mot mémoire qui contient un code indiquant le résultat de l'opération d'entrée sortie sur un fichier.

Exemple : Si l'on veut tester une fin de fichier on pourra utiliser une instruction

SI (ST) ET 64 ALORS...
IF (ST) AND 64 THEN

Il faut alors considérer (ST) ET 64 comme une expression logique qui est vraie si le bit 6 est positionné dans ST ($2^6 = 64$).

Si l'on veut par contre tester l'absence d'erreur il suffit d'utiliser une instruction

IF ST < > 0 THEN...

Remarques. − Ce type de fonctionnement n'est pas standard en BASIC, mais il est bien utile lorsque l'on utilise des cassettes magnétiques dont la fiabilité n'est pas à toute épreuve !

Ce mot d'état, qui s'appelle ST ("Status") sur le "PET", peut prendre différentes valeurs dans les conditions suivantes :

- *blocs trop courts :* si l'on lit des données et que l'on rencontre un délimiteur avant le nombre de caractères attendus (le code est 4) ;
- *blocs trop longs :* si l'on lit des données et que l'on ne rencontre pas de délimiteur quand le nombre de caractères attendu a été lu, le code est alors 8 ;
- *erreur de lecture fatale :* il existe une impossibilité de lire une donnée ; le code est 16 ;

- *erreur de contrôle :* les données ont été lues mais le contrôle effectué sur l'ensemble des données ne concorde pas avec le mot de contrôle lu sur le support (code 32) ; dans ce cas on peut réessayer une lecture ;
- *fin de fichier* (code 64) : il est positionné lorsque la fin du fichier est rencontrée lors d'une lecture ;
- *fin de bande* (code 128) : ceci permet de savoir qu'au-delà il n'y a plus de fichier.

Ce mot d'état peut être testé par programme grâce à l'utilisation des opérateurs logiques ET, OU, PAS.

En effet, une opération logique ET avec le mot d'état permet de savoir si un bit est positionné ou non.

Exemple :

Soit un programme qui rentre des mots et les inscrit sur un fichier appelé FICH. On s'arrête lorsque l'on rencontre le mot FIN.

Ensuite on rebobine la cassette et on fait l'opération inverse de lecture.

L'on obtient le programme suivant :

```
 10    OPEN 1, 1, 2, "FICH"
 20    INPUT Q$
 30    PRINT #1, Q$
 40    IF Q$ < > "FIN" THEN 20
 50    CLOSE 1
 60    INPUT "REBOBINER LA CASSETTE !" ; N
 70    OPEN 1, 1, 0, "FICH"
 80    INPUT#1, Q$
 90    PRINT Q$ ;
100    IF Q$ < > "FIN" THEN 80
110    CLOSE 1
120    END
```

Exécution :

```
?    NOUS ®
?    ECRIVONS®
?    DES ®
?    MOTS ®
?    SUR LE ®
?    FICHIER FICH ®
?    FIN®
REBOBINER LA CASSETTE ! 1®
NOUS ECRIVONS DES MOTS SUR LE FICHIER FICH FIN.
```

6. LE TRAITEMENT DES FICHIERS SUR DISQUES

Nous avons vu que cela nécessitait un support logiciel de type "DOS" que l'on peut appeler le système de gestion de fichiers sur disque (SGFD).

Ce SGFD est accessible, d'une part, à l'aide de commandes système qui font partie du moniteur, d'autre part, à l'aide des primitives de gestion de fichiers que l'on peut utiliser dans un programme.

Ainsi, par exemple, si l'on veut avoir accès à la liste des fichiers sur disque, on utilisera une commande demandant cette liste. Cela peut être une commande telle que :
ANNUAIRE ® (DIRECTORY)
ou CATALOG ®

Par contre, si l'on veut écrire sur un fichier, on utilisera une instruction d'entrée-sortie.

Nous présenterons d'abord les commandes, puis les instructions.

Nous prendrons le cas d'un système de fichiers sur disque souple, et les exemples seront pris sur un système APPLE avec double unité de disques. Il est bien entendu que des commandes similaires doivent exister sur tout système ayant un DOS.

1. Initialisation d'une disquette

Si l'on dispose d'une disquette vierge (venant du fournisseur), il faut au préalable réaliser une opération dite de formattage ou d'initialisation du support. En effet, chaque système dispose d'une technique particulière pour organiser les données sur disque. Cette opération est donc nécessaire même si l'on utilise un standard IBM par exemple.

Cette instruction peut disposer de plusieurs options.

Sur un système APPLE, par exemple, on aura :

INIT NOM FICHIER, n° du lecteur enregistreur, n° de volume, n° de contrôleur.

Le nom de fichier est libre et sert simplement à un fichier qui permet de stocker un programme de démarrage du SGFD (c'est le « bootstrap »).

Les autres paramètres sont :

– le numéro de l'unité de disque utilisée, s'il y en a plusieurs (ce paramètre commence ici par D pour "Drive") ;

– le numéro de volume qui est un numéro assigné à la disquette (de 1 à 254). Ce paramètre commence par la lettre V ;

– le numéro de contrôleur d'entrée-sortie (il peut y en avoir de 1 à 7). S'il est omis, on prend la valeur 1 par défaut (lettre S pour "Slot").

Exemple :

INIT BONJOUR, D1, V2, S1

initialise une disquette avec un fichier initial appelé "BONJOUR" sur l'unité n°1, avec un numéro de volume 2 et le contrôleur n°1.

2. Annuaire d'une disquette

La liste des fichiers disponibles sur la disquette peut être obtenue grâce à une commande :

CATALOG, D n° de l'unité

si le numéro est absent, la valeur par défaut est mise à 1.

Exemple :

CATALOG, D2

visualise ou imprime l'annuaire de la deuxième unité de disque.

En même temps, il peut y avoir visualisation du type du fichier et de la place qu'il occupe sur le disque (nombre de secteurs).

3. Stockage de fichiers programmes

L'une des premières utilités d'un SGFD est de permettre le stockage des programmes sur le disque.

Il existe une commande moniteur qui permet de le faire, c'est-à-dire de sauvegarder le programme actuellement en mémoire centrale sur disque.

C'est la commande :

SAVE NOM DU PROGRAMME, Dn
(SAUVEGARDER)

On précise le nom du programme qui sera alors considéré comme un nom de fichier sur le disque.

Dn est le paramètre numéro d'unité.

Exemple :

SAVE VIE,D2

enregistrera le programme actuellement en mémoire sous le nom de fichier VIE sur le disque n° 2.

Attention : Il faut faire attention aux homonymes ! Si un fichier n'est pas protégé en écriture, il peut être recouvert par un autre fichier de même nom.

4. Lecture en mémoire d'un fichier programme

L'opération inverse qui consiste à lire un fichier programme situé sur disque est également nécessaire.

C'est ce que l'on appelle l'opération de chargement d'un programme en mémoire.

C'est la commande :

LOAD NOM DE FICHIER, Dn
(CHARGER)

La signification des paramètres est la même que ci-dessus.

Si le nom n'existe pas dans le catalogue, il y a envoi d'un message d'erreur.

5. Chargement avec lancement d'un programme

C'est l'opération identique à la précédente, mais avec demande de l'exécution du programme.

On utilise alors la commande

RUN NOM DE FICHIER PROGRAMME, Dn
(MARCHE)

6. Effacement d'un fichier

Lorsque l'on veut détruire un fichier existant, on utilise une commande d'effacement :

DELETE NOM FICHIER, Dn
(EFFACER)

Cette commande efface le nom du fichier dans le catalogue et libère la place occupée par ce fichier, à condition qu'il ne soit pas protégé.

Cette commande est valable quel que soit le type du fichier.

7. Protection et déprotection des fichiers.

Afin d'éviter les destructions accidentelles ou intempestives de fichiers, il est prudent de protéger ses fichiers. Cette opération se fait à l'aide de la commande :

LOCK NOM FICHIER, Dn
(VERROUILLER)

Ainsi, toute opération de réécriture ou d'effacement de ce fichier sera interdite.

Si l'on veut par contre réaliser l'opération inverse on utilisera la commande :

UNLOCK NOM FICHIER, Dn
(DEVERROUILLER)

Le fichier redevient alors « vulnérable ».

Remarque. – Si l'on veut protéger le contenu d'un disque complet, il vaudra mieux avoir recours à un mécanisme de protection écriture physique. Cela est réalisé grâce à une encoche de protection écriture sur l'enveloppe de la disquette.

Inversement, si l'on veut déprotéger une disquette, on peut le faire en remettant un onglet à l'endroit où se trouve l'encoche.

8. Changement d'un nom de fichier

Dans certains cas, on peut avoir besoin de renommer un fichier. Ceci peut se faire grâce à la commande :

RENAME ANCIEN NOM, NOUVEAU NOM, Dn
(RENOMMER)

Cette opération met à jour le catalogue en remplaçant l'ancien nom par le nouveau, sans que le contenu du fichier soit modifié.

9. Utilisation de commandes dans un programme BASIC

Toutes ces commandes sont également accessibles par l'intermédiaire de l'interpréteur BASIC sur le système APPLE.

Pour cela, il faut utiliser des instructions de sortie (PRINT) dont le premier caractère est un caractère de contrôle (CONTROLE D noté D^c). Au clavier, ce caractère est obtenu en appuyant sur les touches CONTROL et D simultanément. Le code ASCII de ce caractère est le nombre 4.

Le caractère D^c peut donc être défini par :

D$ = CHR$(4)

c'est-à-dire le caractère ayant pour code ASCII : 4. Ce caractère doit être suivi par la chaine de caractères correspondant à la commande que l'on veut effectuer.

Exemple :

```
10    D$ = CHR$(4)
20    PRINT D$ ; "CATALOG"
```

a pour effet d'imprimer ou de visualiser le catalogue.

LE TRAITEMENT DES FICHIERS SEQUENTIELS SUR DISQUE

Maintenant nous considérons des fichiers de données qui sont traitées par des programmes en BASIC.

Ces traitements font appel au SGFD (DOS) par l'intermédiaire de primitives ou instructions au SGFD.

Ces primitives ne sont pas accessibles en dehors d'un programme. Quel que soit le système, les fonctions de ces primitives sont toujours les mêmes :

l'OUVERTURE d'un fichier	(OPEN)
la FERMETURE	(CLOSE)
la LECTURE	(READ)
l'ECRITURE	(WRITE)
l'AJOUT	(APPEND)
le POSITIONNEMENT	(POSITION)

Pour définir ces primitives, on a le choix entre l'utilisation de nouveaux mots clés (OPEN...READ, WRITE...) et l'introduction de nouvelles instructions BASIC, ou bien l'on peut, utiliser les instructions d'entrées-sorties standards en distinguant les primitives au SGFD par un caractère spécial.

Sur la plupart des systèmes, c'est la première solution qui a été choisie, mais il n'y a aucun standard défini. La seconde solution est celle qui est utilisée sur le système APPLE : elle présente l'avantage de définir des programmes syntaxiquement standards. C'est celle que nous utiliserons ci-dessous.

Définition d'une primitive

La méthode utilisée est identique à celle déjà vue pour la définition de commandes au SGFD en BASIC.

Le caractère spécial utilisé est le caractère CONTROLE D (D^c), qui a pour code ASCII 4.

De manière générale, si l'on a défini : D$ = CHR$ (4), une primitive au SGFD est définie par l'instruction :

PRINT D$; "PRIMITIVE"

Le texte de la primitive est exprimé entre guillemets et varie suivant la nature de celle-ci. On peut aussi définir une variable PR$ telle que

PR$ = "TEXTE DE LA PRIMITIVE"

puis l'instruction PRINT D$; PR$

Structure d'un fichier séquentiel en BASIC

En BASIC, un fichier est une suite de caractères (codés en ASCII). Ces caractères sont organisés en articles, champs ou items, séparés par des caractères retour chariot ® dont le code ASCII est 13.

Exemple :

Soit un fichier contenant des adresses de personnes. Dans ce fichier, on a par exemple la suite de caractères :

DUPONT JEAN ® 24, RUE LONGUE ® 75010 ® PARIS ®

Cette suite de caractères est enregistrée séquentiellement dans le fichier, chaque caractère étant codé en ASCII. Le retour chariot ® sert de séparateur entre les différents items.

LES DIFFERENTES PRIMITIVES D'UN SGFD SEQUENTIEL :

L'ouverture d'un fichier

La primitive correspondante est :

OUVRIR $ = "OPEN NOM FICHIER, Dn, Vm, Sp"
(OUVRIR)

L'instruction est alors :

PRINT D$; OUVRIR$

Cette instruction a pour effet de se positionner au début du fichier dont le nom est spécifié et qui se trouve sur le disque Dn, le Volume Vm et associé au contrôleur Sp.

En même temps, le système prévoit une zone de travail en mémoire centrale pour effectuer des lecture/écriture de fichier.

Si le fichier n'existe pas, il y a un message d'erreur.

S'il est déjà ouvert, l'instruction a pour effet de le fermer puis de le réouvrir.

Remarque. – Le nombre de fichiers que l'on peut ouvrir simultanément est limité. Il est précisé par la commande MAXFILES, qui permet d'avoir jusqu'à 16 fichiers ouverts en même temps. L'option par défaut est de 3 fichiers simultanés.

Exemple :

```
10    D$ = CHR$(4)
20    PRINT D$ ; "OPEN ADRESSE"
```

réalise l'ouverture du fichier ADRESSE. Les options par défaut sont D1 et V1.

Fermeture d'un fichier

Cette opération est l'opération inverse de la précédente, car elle libère la place mémoire utilisée lors de l'ouverture. D'autre part, si

le fichier avait été utilisé en écriture, cela permet d'écrire la fin du fichier. L'oubli de fermeture d'un fichier peut donc résulter en une perte de données.

La syntaxe en est :

CLOSE NOM DE FICHIER
(FERMER)

Il n'est pas ici nécessaire de préciser les autres paramètres, car le fichier a été ouvert préalablement.

La commande CLOSE sans nom de fichier permet de fermer tous les fichiers.

Exemple :

```
100       PRINT D$ ; "CLOSE ADRESSE"
```

permet de fermer le fichier adresse.

Ecriture dans un fichier séquentiel

Le principe d'écriture est le même que pour un fichier sur cassette. On écrit des items séparés par des caractères retour chariot ®. L'écriture suppose que le fichier ait été ouvert.

La primitive permettant d'initialiser une écriture est :

WRITE NOM DE FICHIER
(ECRIRE)

mais cette primitive n'a pas pour rôle d'écrire véritablement dans le fichier, elle indique seulement que toutes les instructions de sortie suivant cette instruction dans le cours du programme seront faites dans ce fichier.

Exemple :

```
  1     REM ECRITURE D'ADRESSES
  5     DIM N$(100),AD$(100)
 10     D$ = CHR$(4)
 20     PRINT D$;"OPEN ADRESSE"
 30     ECRI$ = "WRITE ADRESSE"
 35     REM
 36     REM ENTREE DES ADRESSES
 37     REM
 40     FOR I = 1 TO 100
 50     INPUT "NOM PRENOM";N$(I)
 55     IF N$(I) = "ZZ" THEN 80
 60     INPUT "ADRESSE";AD$(I)
 70     NEXT I
 74     REM
```

```
 75    REM ECRITURE SUR DISQUE
 76    REM
 80    PRINT D$;ECRI$
 90    FOR J = 1 TO I
100    PRINT N$(J)
105    PRINT AD$(J)
110    NEXT J
120    PRINT D$;"CLOSE ADRESSE"
130    END
```

Exécution :

```
]RUN
NOM PRENOM ARTHUR RIMBAUD
ADRESSE CHARLEROI
NOM PRENOM STEPHANE MALLARME
ADRESSE PARIS
NOM PRENOM VERLAINE PAUL
ADRESSE PARIS
NOM PRENOM TRISTAN CORBIERE
ADRESSE MORLAIX
NOM PRENOM SAINT POL ROUX
ADRESSE CROZON
NOM PRENOM ZZ
```

Ce programme permet de rentrer au clavier des adresses sous la forme d'un nom et prénom puis d'une adresse, le tout est écrit sur le fichier adresse.

Si l'on rencontre un nom = ZZ, on arrête la lecture des données, puis on écrit dans le fichier.

Lecture d'un fichier séquentiel

La lecture suppose que le fichier ait été ouvert. Là aussi, la primitive de lecture indique que toutes les instructions d'entrée (INPUT) qui suivent se font à partir du fichier désigné par la primitive de lecture dont la syntaxe est :

READ NOM DE FICHIER
(LIRE)

Exemple :

Soit à lire le fichier adresse écrit précédemment

```
  1    REM LECTURE D'ADRESSES
  5    DIM N$(100),AD$(100)
 10    D$ = CHR$(4)
 20    PRINT D$ ;"OPEN ADRESSE"
```

```
 30        LIRE$ = "READ ADRESSE"
 34        PRINT D$;LIRE$
 35        REM
 36        REM LECTURE SUR DISQUE
 37        REM
 40        FOR I = 1 TO 100
 50        INPUT N$(I)
 55        IF N$(I) = "ZZ" THEN 80
 60        INPUT AD$(I)
 70        NEXT I
 74        REM
 75        REM IMPRESSION DES ADRESSES
 76        REM
 80        I = I - 1
 90        FOR J = 1 TO I
100        PRINT N$(J)
105        PRINT AD$(J)
106        PRINT
110        NEXT J
120        PRINT D$;"CLOSE ADRESSE"
130        END
```

Exécution :

```
]RUN
ARTHUR RIMBAUD
CHARLEROI
SPEPHANE MALLARME
PARIS
VERLAINE PAUL
PARIS
TRISTAN CORBIERE
MORLAIX
SAINT POL ROUX
CROZON
```

Là aussi, le test fin de fichier est fait sur le nom : "zz". Cependant ceci n'est pas très pratique lorsque l'on veut ajouter des noms au fichier. On verra ci-dessous qu'il vaut mieux tester la fin de fichier physique.

Ajout dans un fichier séquentiel

Cette opération consiste à écrire de nouvelles données à la fin d'un fichier déjà existant. Pour cela, on utilise la primitive AJOUT (APPEND).

Là aussi, cette primitive n'a pas pour effet d'écrire réellement, mais d'indiquer que toutes les opérations de sortie (PRINT) qui suivent devront être écrites à la fin du fichier spécifié. L'effet de la

primitive est donc simplement de se positionner à la fin du fichier. La primitive AJOUT doit être suivie d'une primitive d'écriture.

Exemple :

Soit le fichier adresse déjà créé, si l'on veut pouvoir rajouter des adresses on définit la primitive :

AJOUT$ = "APPEND ADRESSE"

et l'opération d'ouverture est remplacée par l'instruction :

PRINT D$; AJOUT$

Avant de présenter le programme, il est nécessaire de modifier le programme d'écriture initial de façon à ne pas écrire le nom "zz". On obtient alors :

```
1    REM ECRITURE D'ADRESSES
5    DIM N$(100),AD$(100)
10   D$ = CHR$(4)
20   PRINT D$;"OPEN ADRESSE"
30   ECRI$ = "WRITE ADRESSE"
35   REM
36   REM ENTREE DES ADRESSES
37   REM
40   FOR I = 1 TO 100
50   INPUT "NOM PRENOM";N$(I)
55   IF N$(I) = "ZZ" THEN 80
60   INPUT "ADRESSE";AD$(I)
65   PRINT N$
70   NEXT I
74   REM
75   REM ECRITURE SUR DISQUE
76   REM
80   PRINT D$;ECRI$
90   FOR J = 1 TO I - 1
100  PRINT N$(J)
105  PRINT AD$(J)
110  NEXT J
120  PRINT D$;"CLOSE ADRESSE"
130  END
```

Le programme d'AJOUT est alors :

```
1    REM AJOUT D'ADRESSE DANS UN FICHIER SEQUENTIEL
5    DIM N$(100),AD$(100)
10   D$ = CHR$(4)
20   AJOUT$ = "APPEND ADRESSE"
25   PRINT D$;AJOUT$
30   ECRI$ = "WRITE ADRESSE"
35   REM
36   REM ENTREE DES ADRESSES
37   REM
40   FOR I = 1 TO 100
```

```
 50    INPUT "NOM PRENOM";N$(I)
 55    IF N$(I) = "ZZ" THEN 80
 60    INPUT "ADRESSE";AD$(I)
 65    PRINT N$
 70    NEXT I
 74    REM
 75    REM ECRITURE SUR DISQUE
 76    REM
 80    PRINT D$;ECRI$
 90    FOR J = 1 TO I - 1
100    PRINT N$(J)
105    PRINT AD$(J)
110    NEXT J
120    PRINT D$;"CLOSE ADRESSE"
130    END
```

Exemple d'exécution :

```
]RUN
NOM PRENOM MAX JACOB
ADRESSE QUIMPER
NOM PRENOM ANDRE BRETON
ADRESSE PARIS
NOM PRENOM ZZ
```

Les deux noms précédents ont été ajoutés à la fin du fichier.

Le programme de lecture modifié teste la fin de fichier à l'aide de l'instruction ONERR GOTO 80. Si le code obtenu par une instruction PEEK (222) est égal à 5, c'est une fin de fichier. On obtient alors :

```
  1    REM LECTURE D'ADRESSES
  5    DIM N$(100),AD$(100)
 10    D$ = CHR$(4)
 20    PRINT D$;"OPEN ADRESSE"
 30    LIRE$ = "READ ADRESSE"
 34    PRINT D$;LIRE$
 35    REM
 36    REM LECTURE SUR DISQUE
 37    REM
 40    FOR I = 1 TO 100
 45    ONERR GOTO 80
 50    INPUT N$(I)
 60    INPUT AD$(I)
 70    NEXT I
 74    REM
 75    REM IMPRESSION DES ADRESSES
 76    REM
 80    C = PEEK (222)
 85    IF C = 5 THEN 90
 86    PRINT "ERREUR DOS NO" ; C
```

```
 90     FOR J = 1 TO I - 1
100     PRINT N$(J)
105     PRINT AD$(J)
106     PRINT
110     NEXT J
120     PRINT D$;"CLOSE ADRESSE"
130     END
```

Exécution :

```
]RUN
ARTHUR RIMBAUD
CHARLEROI
STEPHANE MALLARME
PARIS
VERLAINE PAUL
PARIS
TRISTAN CORBIERE
MORLAIX
SAINT POL ROUX
CAMARET
MAX JACOB
QUIMPER
ANDRE BRETON
PARIS
```

On vérifie que les deux derniers noms ont bien été ajoutés à la fin du fichier.

Positionnement dans un fichier séquentiel

Cette primitive permet de se positionner n'importe où dans un fichier en vue d'une lecture ou d'une écriture. La position est repérée par un déplacement relatif à la position actuelle d'un pointeur repérant l'item courant. Elle permet de se déplacer de K items vers l'avant.

La syntaxe de la primitive est :

POSITION NOM DE FICHIER, R K

R est l'initiale du mot Record (enregistrement), tandis que K est un entier précisant le nombre d'items à sauter.

Ainsi, par exemple :
Si K = 0 (ou absent) c'est l'item courant qui sera lu ou écrit.
Si K = 1, c'est l'item suivant qui sera lu, ou écrit..., etc.
Si K = *n*, c'est le nième item en avant de l'item courant qui sera traité.

Si la valeur de K est telle qu'il n'y a plus d'items, dans le fichier, alors, on obtient un message d'erreur.

L'instruction de positionnement doit précéder l'instruction de lecture ou écriture du fichier.

Exemple :

Dans le fichier ADRESSE précédent, écrire un programme qui lise seulement les noms des individus en ignorant la partie adresse.

On définit une primitive AVANCE$ et dans l'instruction 60 on spécifie un positionnement de 1, c'est-à-dire un saut d'un enregistrement. Le programme suivant est alors obtenu :

```
  1   REM LECTURE D'ADRESSES
  5   DIM N$(100),AD$(100)
 10   D$ = CHR$(4)
 20   PRINT D$;"OPEN ADRESSE"
 25   AVANCE$ = "POSITION ADRESSE,R"
 30   LIRE$ = "READ ADRESSE"
 35   REM
 36   REM LECTURE SUR DISQUE
 37   REM
 40   FOR I = 1 TO 100
 45   ONERR GOTO 80
 46   PRINT D$;LIRE$
 50   INPUT N$(I)
 60   PRINT D$;AVANCE$;1
 70   NEXT I
 74   REM
 75   REM IMPRESSION DES ADRESSES
 76   REM
 80   C = PEEK (222)
 85   IF C = 5 THEN 90
 86   PRINT "ERREUR DOS NO";C
 90   FOR J = 1 TO I - 1
100   PRINT N$(J)
106   PRINT
110   NEXT J
120   PRINT D$;"CLOSE ADRESSE"
130   END
]RUN
ARTHUR RIMBAUD
STEPHANE MALLARME
VERLAINE PAUL
TRISTAN CORBIERE
SAINT POL ROUX
MAX JACOB
ANDRE BRETON
```

Utilisation d'un fichier en mode caractère

Jusqu'à présent nous avons vu qu'un fichier séquentiel était constitué d'items séparés par des caractères Retour à la ligne ®.

La primitive POSITION permet de se positionner au début des items et le paramètre précisé dans cette primitive permet de sauter un certain nombre d'items.

Ceci n'est pas toujours suffisamment souple, notamment si l'on veut traiter le fichier en mode caractère.

Dans ce cas, il faut pouvoir disposer d'un mécanisme permettant de se positionner n'importe où dans le fichier. Ceci est possible grâce à l'utilisation d'un système de pointeur qui indique l'emplacement exact où l'on veut lire ou écrire dans un fichier.

Dans ce cas, les primitives WRITE et READ doivent être associées à un pointeur indiquant la position exacte par rapport au début du fichier de l'emplacement où l'on veut écrire.

La syntaxe de ces primitives est alors :

WRITE NOM DE FICHIER, B_n
(ECRIRE)

B est le caractère spécifiant le pointeur de caractère (B pour "Byte") et *n* est la position exacte en nombre de caractères par rapport au début du fichier où l'on veut se positionner en écriture.

Le début du fichier correspond à la position 0.

I	Caractère 0
T	– 1
E	– 2
M	– 3
1	– 4
®	– 5
I	– 6
T	– 7
E	– 8
M	– 9
2	– 10
®	– 11
I	– 12
T	– 13

E	– 14
M	– 15
3	– 16
®	– 17
FIN FICHIER	

Exemple :

WRITE FICH, B 14

indique que l'on veut écrire dans le fichier nommé FICH, à partir du 15ᵉ caractère.

Remarques. – La valeur du pointeur est une valeur absolue qui doit tenir compte des caractères Retour à la ligne ®, des blancs et de tout autre caractère présent physiquement dans le fichier.

Il faut donc faire très attention avec une telle primitive, car l'on peut de ce fait écraser des informations au-delà d'un item donné et même au-delà du fichier considéré si le pointeur fait référence à un caractère se trouvant au-delà de la fin du fichier.

Nous conseillons donc au débutant d'être très prudent avec cette instruction.

Il est d'ailleurs bien préférable de se familiariser avec ce type d'instruction en utilisant la primitive de lecture dont la syntaxe est similaire :

READ NOM FICH, B_n
(LIRE)

Les paramètres ont la même signification que pour la primitive WRITE.

Exemple :

READ FICH, B52

permet de se positionner au 53ᵉ caractère du fichier FICH pour la lecture.

Là aussi, la primitive LIRE n'a pas pour effet de lire et doit donc être suivie d'instruction d'entrée (INPUT) qui lira les données à partir de la position définie par le pointeur.

Application : création d'un système de fichiers indexés

La possibilité de travailler en mode caractère permet de créer un système de fichiers indexés utilisant les primitives du SFGD.

Organisation d'un tel fichier

Un fichier indexé est un fichier qui est décomposé en articles qui sont écrits séquentiellement sur le disque, mais auquel on peut avoir accès par l'intermédiaire d'un index.

Un index est une table qui est balayée séquentiellement et qui indique l'emplacement de chaque article dans le fichier.

On peut le schématiser par une suite de doublets :

NOM ARTICLE ADR. ARTICLE

Le nom d'article peut éventuellement être un numéro et l'adresse de l'article est la position en nombre de caractères (octets) relative au début du fichier.

Si l'index est de longueur fixe, on peut le mettre au début de fichier.

Si l'index est de longueur variable, on doit le mettre à la fin du fichier. Cette solution est préférable, car plus générale, mais elle implique une grande sécurité de fonctionnement, car, si l'on ne peut pas réécrire l'index ou s'il est mal réécrit, le fichier est difficilement récupérable.

La structure d'un fichier indexé peut alors être représentée de la façon suivante :

ADR INDEX	
1er article	
2e article	
...	
nième article	
N° ART AD	
N° ART AD	
	INDEX
N° ART AD	

Shéma d'un fichier indexé

Remarque. – Si l'index contient les noms d'articles ou numéros, les articles ne sont pas obligatoirement écrits dans l'ordre 1er, 2e..., mais au fur et à mesure des besoins.

Les primitives associées

Les primitives n'existent pas sur les SGFD des micro-ordinateurs pris en exemple. Cependant, on peut les développer grâce à la possibilité de définir des pointeurs.

Du point de vue du SGFD, les fichiers correspondants seront des fichiers séquentiels.

L'index devra donc être maintenu par programme.

Les primitives que l'on devra développer seront donc :

- la CREATION d'un index ;
- l'ECRITURE d'un article ;
- la LECTURE d'un article.

La lecture d'un article impliquant la lecture de l'index en mémoire centrale, on peut également prévoir une primitive d'OUVERTURE d'un fichier indexé qui consistera à lire l'index en mémoire.

De même, l'écriture d'un article impliquant une mise à jour et une écriture de l'index, on peut introduire une primitive de FERMETURE d'un fichier indexé qui aura pour effet d'écrire l'index.

Si le fichier est évolutif, on peut également prévoir la primitive d'EFFACEMENT d'un article.

Quant à l'effacement d'un fichier, il n'est pas nécessaire de le prévoir puisque l'effacement d'un fichier séquentiel existe déjà.

Ces primitives sont laissées à la programmation du lecteur à titre d'exercice.

LES FICHIERS EN ACCES DIRECT (ALEATOIRE)

Une autre méthode d'accès est possible, c'est la méthode d'accès direct (appelée aussi aléatoire bien que cela soit quelque peu impropre).

Dans ce cas, le fichier est organisé en articles ou en enregistrements, comme pour l'accès indexé.

Cependant, ici, il n'y a pas d'index à proprement parler, et c'est le numéro d'enregistrement qui permet d'accéder directement à un article.

Ceci implique donc que les articles soient de longueur fixe et que la place soit réservée pour les articles qui ne sont pas encore écrits.

L'accès est appelé direct car l'emplacement d'un article peut être obtenu directement par un simple calcul :

ADRESSE DEBUT = N° ARTICLE × LONGUEUR EN CARACTERES

L'avantage d'une telle méthode est la rapidité d'accès ; l'inconvénient est la perte de place occasionnée par les articles non encore écrits et éventuellement par la rigidité imposée par la longueur fixée des articles.

Les primitives d'un SGFD en accès direct

On retrouve certaines primitives du SGFD séquentiel, mais avec des paramètres supplémentaires.

OUVERTURE en accès direct

Le paramètre supplémentaire qui doit être spécifié est la longueur de l'article ou enregistrement.

La syntaxe de la primitive est alors :

OPEN NOM FICHIER, L*n*
(OUVRIR)

L définit le paramètre longueur et *n* est un nombre entier de 1 à 32 767, qui représente le nombre de caractères fixé pour un article.

Remarques. – En ouverture on peut également préciser les paramètres habituels définissant le disque, le volume et le contrôleur (cf. fichier séquentiel).

La valeur prise par défaut pour L est 1. Il faut donc connaître la longueur des articles pour utiliser un fichier en accès direct.

Exemple :
```
10    D$ = CHR$ (4)
20    PRINT D$ "OPEN ADRESSE, L100"
......
```
permet d'ouvrir un fichier adresse ayant des enregistrements de 100 caractères, ce qui correspond à 5 lignes de 20 caractères pour une adresse.

LECTURE/ECRITURE en accès direct

Ici, il faut préciser un paramètre nécessaire à l'accès direct, à savoir le numéro de l'article ou enregistrement. On a donc les primitives :

READ NOM FICHIER, R*n*, B*n*
(LIRE)
WRITE NOM FICHIER, R*n*, B*n*
(ECRIRE)

R*n* est le paramètre qui précise l'article (Record) à lire ou écrire, *n* est le numéro de l'article à lire ou écrire. B*n* est un paramètre facultatif qui a déjà été vu pour les fichiers séquentiels, mais ici il

représente le nième caractère à partir duquel il faudra lire ou écrire dans l'article spécifié. C'est donc un pointeur à l'intérieur de l'article. Il faut l'utiliser avec précaution surtout en écriture.

Application. – Soit à créer un fichier d'adresses d'individus.

On suppose également que ces individus sont identifiés par un numéro unique. On limite la taille d'un article adresse à 100 caractères.

Pour préciser les primitives de lecture/écriture d'un article n, il suffit de définir une chaîne de caractères telle que :

"WRITE NOMFICH, R"

et de lui associer un entier N qui suivra immédiatement cette chaîne de caractères dans l'instruction correspondante, soit par exemple :

PRINT D$; "WRITE NOMFICH, R" ; N

indique que l'on veut écrire l'enregistrement N du fichier appelé NOMFICH.

On obtient alors le programme suivant qui commence par écrire des adresses et permet ensuite de les relire en précisant le numéro de l'enregistrement ou article.

Programme d'écriture

```
  1   REM FICHIER ADRESSE EN ACCES DIRECT
  4   DIM N(100)
  5   DIM N$(100),AD$(100)
 10   D$ = CHR$(4)
 15   OUVR$ = "OPEN ADIRECT,L100"
 16   FERME$ = "CLOSE ADIRECT"
 20   PRINT D$;OUVR$
 30   ECRI$ = "WRITE ADIRECT,R"
 35   REM
 36   REM ENTREE DES ADRESSES
 37   REM
 40   FOR I = 1 TO 100
 45   INPUT "NUMERO";N(I)
 50   INPUT "NOM PRENOM";N$(I)
 55   IF N$(I) = "ZZ" THEN 80
 60   INPUT "ADRESSE";AD$(I)
 65   PRINT N$
 70   NEXT I
 74   REM
 75   REM ECRITURE SUR DISQUE
 80   REM
 90   FOR J = 1 TO I - 1
 95   PRINT D$;ECRI$;N(J)
100   PRINT N$(J)
105   PRINT AD$(J)
```

```
110     NEXT J
120     PRINT D$;FERME$
130     END
```

```
RUN
NUMERO 1
NOM PRENOM BAUDELAIRE CHARLE
ADRESSE PARIS
NUMERO 1
NOM PRENOM BAUDELAIRE CHARLES
ADRESSE PARIS
NUMERO 4
NOM PRENOM CHATEAUBRIAND
ADRESSE RENNES
NUMERO 9
NOM PRENOM PREVERT MARCEL
ADRESSE PARIS
NUMERO 17
NOM PRENOM ZZ
```

Le programme suivant est un programme permettant de lire un article du fichier, de le corriger et le réécrire :

```
  1     REM FICHIER ADRESSE EN ACCES DIRECT
  2     REM
  3     REM PRIMITIVES DU SGFD
  4     REM
 10     D$ = CHR$(4)
 15     OUVR$ = "OPEN ADIRECT,L100"
 16     FERME$ = "CLOSE ADIRECT"
 20     ECRI$ = "WRITE ADIRECT,R"
 30     LIRE$ = "READ ADIRECT,R"
 35     REM
 36     REM ENTREE DU NUMERO D'ARTICLE K
 37     REM
 40     INPUT "NUMERO";K%
 45     IF K% < = 0 THEN 230
 50     ONERR GOTO 190
 52     PRINT D$;OUVR$
 55     PRINT D$;LIRE$;K%
 56     REM
 57     REM LECTURE DE L'ARTICLE SUR DISQUE
 58     REM
 60     INPUT N$: INPUT AD$
 65     REM
 66     REM IMPRESSION DE L'ADRESSE
 67     REM
 70     PRINT : PRINT N$: PRINT AD$
 72     PRINT
 75     PRINT D$;FERME$
```

```
 76   REM
 77   REM CORRECTION DE L'ARTICLE
 78   REM
 80   INPUT "REECRITURE O/N?";R$
 90   IF R$ = "0" THEN 130
 95   GOTO 40
100   REM
110   REM ECRITURE D'UN ARTICLE
120   REM
130   INPUT "NOM PRENOM";N$
140   INPUT "ADRESSE";AD$
150   PRINT D$;OUVR$
160   PRINT D$;ECRI$;K%
170   PRINT N$: PRINT AD$
180   PRINT D$;FERME$
185   GOTO 220
186   REM
187   REM CAS D'ERREUR TESTER LA FIN DE FICHIER OU
      L'ABSENCE D'ARTICLE
188   REM
190   C = PEEK (222)
200   IF C = 5 THEN INPUT "ECRITURE O/N?" ; R$: GOTO 90
210   PRINT "ERREUR DOS NO";C
220   GOTO 40
230   END
```

Exercices

1. *Ecrire un programme qui permette de créer un index dans un fichier séquentiel.*
   Pour cela, le premier enregistrement d'un fichier séquentiel sera considéré comme un nombre de 5 chiffres (0 à 99 999) qui indiquera l'adresse de l'index.
   Soit

×	
×	
×	
×	ADRESSE
×	INDEX
®	

L'index est composé lui-même de plusieurs enregistrements dont le premier représente le nombre d'articles du fichier sur 3 chiffres (de 0 à 999)

INDEX		
	N	
	N	NOMBRES D'ARTICLES
	N	

2. *Ecrire un programme qui écrive un nouvel article dans un fichier indexé (organisé séquentiellement) ; chaque entrée d'un article dans l'index est composée d'un nom d'article sur 3 caractères (ce peut être un numéro de 3 chiffres) suivi de l'adresse de début de l'article sur 5 chiffres.*

N	
O	NOM D'ARTICLE
M	
A	
D	ADRESSE DE L'ARTICLE
R	
E	
S	
®	

3. *Ecrire un programme qui lise un article d'un fichier indexé.*

CONCLUSION

Dans ce chapitre nous avons présenté les concepts fondamentaux nécessaires au traitement de données structurées dans des fichiers. Bien que les programmes présentés soient simples, ils représentent les différents modes de traitement disponibles sur tous les systèmes. Les primitives présentées sont similaires sur tous les systèmes, y compris les plus gros systèmes. Certes, les systèmes de fichiers doivent être plus élaborés si l'on veut prendre en compte les notions de partage de fichiers entre utilisateurs ainsi que le développement des systèmes de base de données.

Néanmoins, les outils disponibles actuellement sur les plus petits systèmes à disques souples sont suffisamment élaborés pour permettre à des utilisateurs de développer des systèmes de stockage

de données structurées sous forme de petites bases de données, car, dans ce cas, ne se posent pas les problèmes de partage simultané des fichiers ou de la base de données. Le développement des techniques des systèmes répartis permettra dans un avenir proche de rendre accessibles les données stockées sur les plus petits systèmes à partir de n'importe quel terminal ou système connecté à un réseau.

CHAPITRE VI

LE TRAITEMENT GRAPHIQUE EN BASIC

Dans ce chapitre, nous étudierons les possibilités de faire du graphique en BASIC. Bien que cela ne fasse pas partie du langage standard, il existe de nombreux systèmes BASIC qui proposent des instructions permettant de traiter des problèmes graphiques. Il y a encore quelques années, cela nécessitait l'achat de consoles de visualisation graphiques qui étaient relativement coûteuses. Maintenant, des possibilités similaires existent sur les systèmes connectés à des appareils de télévision standard. En particulier, les programmes présentés dans ce chapitre ont été testés sur un système APPLE qui permet également de travailler avec de la couleur.

Nous verrons que les possibilités graphiques apportent une autre dimension au langage. On peut même dire qu'il est dommage que les instructions graphiques ne soient pas mieux standardisées, car, d'un point de vue pédagogique, il est souvent facile et plus agréable de faire comprendre les concepts de la programmation avec des programmes qui visualisent ce qui se passe.

Par-delà les particularités du système utilisé, nous insistons sur les problèmes généraux rencontrés en matière de graphique : changement d'origine, changement d'échelle, traitement des vecteurs.

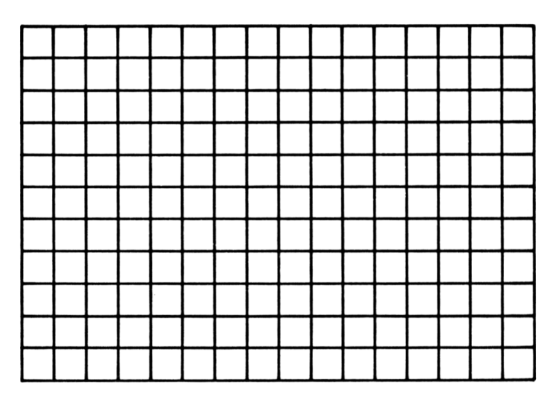
De cette façon, un utilisateur qui ne dispose pas du même système peut sans difficulté l'appliquer à un autre système.

1. INTRODUCTION AU GRAPHIQUE EN BASIC

Bien que le langage BASIC standard ne dispose pas d'instructions graphiques, il existe plusieurs systèmes micro-ordinateurs qui proposent de telles facilités, notamment lorsque le micro-ordinateur est relié à un poste de télévision en couleurs. Ainsi en est-il du système "APPLE", que nous prendrons comme exemple pour la suite de ce chapitre. C'est en effet le système qui nous semble le mieux conçu pour le graphique en utilisant un téléviseur standard.

Le passage en mode graphique

Le mode graphique suppose que l'écran ne soit plus décomposé en lignes de caractères, mais en une grille de points que l'on adresse par un système de coordonnées cartésiennes : l'abscisse (X) et l'ordonnée (Y).



Grille de points sur un écran.

L'utilisation de l'écran comme grille de points (en fait ce sont des carrés) est spécifiée par une commande moniteur.

Sur le système APPLE il existe deux modes correspondant à des grilles différentes et à des possibilités de couleurs différentes.

Le premier mode est le mode graphique simple, utilisant une grille de 40 sur 40 et la possibilité de seize couleurs différentes. Ce mode spécifié par la commande :

GR (Graphique)

Dans ce mode, le bas de l'écran est disponible pour la visualisation de quatre lignes de texte. Le retour à l'utilisation de la totalité de l'écran pour visualiser du texte se fait par la commande : TEXT. Il existe également une commande pour utiliser tout l'écran en mode graphique.

Ces commandes peuvent d'ailleurs être insérées dans un programme BASIC.

Le deuxième mode est le mode graphique de *haute résolution,* utilisant soit une grille de 280 sur 160 avec la partie de l'écran réservée au texte, soit une grille de 280 sur 192, qui correspond à tout l'écran. Dans le premier cas, on utilise une commande :

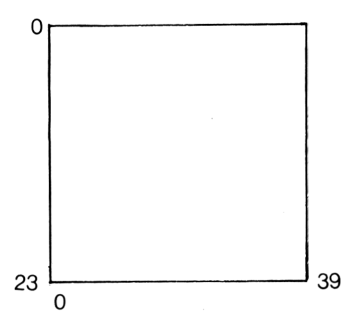
HGR (graphique haute résolution)

Dans le deuxième cas, on utilise la commande :

HGR2 (graphique haute résolution sur tout l'écran)

2. LE MODE GRAPHIQUE SIMPLE

La grille utilisée est de 40 × 40, et chaque point est repéré par des coordonnées allant de 0 à 39. Les points ont des coordonnées allant de 0 à 39. Le point de coordonnées (0,0) est le point le plus en haut et à gauche de l'écran.



On remarquera que si les abscisses vont bien en croissant de la gauche vers la droite, les ordonnées vont en croissant du haut vers le bas, ce qui est contraire à l'usage habituel en mathématique.

LA COULEUR

Avant de voir comment l'on peut visualiser un point sur un écran, il faut définir le jeu des couleurs disponibles. Sur le système APPLE, les couleurs disponibles sont définies par un code numérique allant de 0 à 15 :

0	Noir	8	Brun
1	Brun foncé	9	Orange
2	Bleu foncé	10	Gris
3	Violet lavande	11	Rose
4	Vert foncé	12	Vert
5	Gris foncé	13	Jaune
6	Bleu	14	Bleu aqua
7	Bleu ciel	15	Blanc

La définition d'une couleur se fait à l'aide de l'instruction COULEUR (COLOR) :

COLOR = n $\quad\quad 0 \leqslant n \leqslant 15$

Il est également possible de définir la couleur par une expression arithmétique : COLOR = Expression arithmétique. Dans ce cas, la couleur associée sera obtenue par la valeur correspondant à la partie entière de l'expression modulo 16.

Remarque : Les couleurs peuvent varier d'un téléviseur à un autre et suivant l'interface utilisé.

Visualisation d'un point

Il s'agit d'une commande de sortie précisant les coordonnées du point à visualiser. Cette instruction se retrouve sur plusieurs systèmes BASIC dans des versions similaires. C'est l'instruction DESSINER (PLOT).

La syntaxe en est :

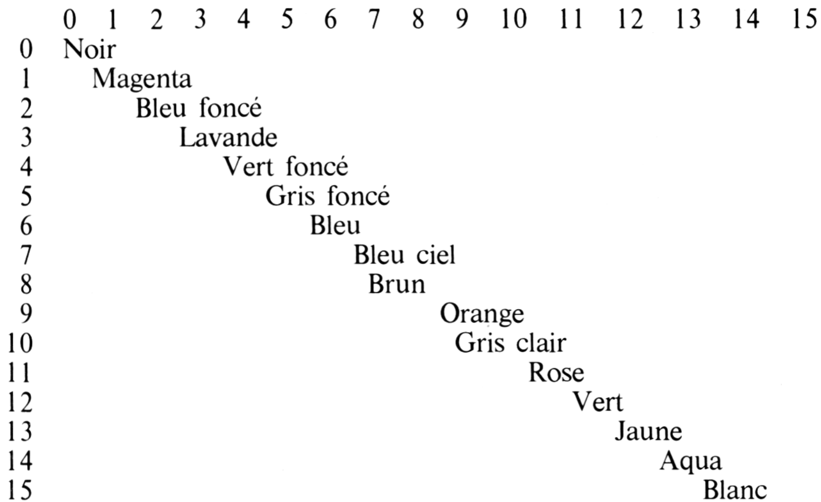
PLOT X, Y

X et Y sont des variables ou des constantes représentant les coordonnées du point à visualiser. La couleur du point sera déterminée par la dernière valeur définie par la commande COLOR. S'il n'y en a pas, la valeur prise par défaut sera le 0 (noir).

Exemple de programme de visualisation des couleurs

```
10    GR
20    FOR I = 0 TO 15
30    COLOR = I
40    PLOT I, I
50    NEXT I
60    END
```

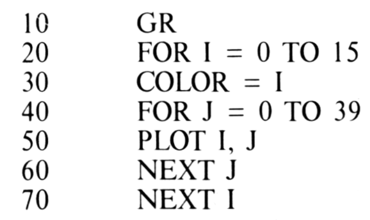
Ce programme visualise les 16 couleurs sur les 16 premiers points de la diagonale principale de la grille :



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	Noir															
1		Magenta														
2			Bleu foncé													
3				Lavande												
4					Vert foncé											
5						Gris foncé										
6							Bleu									
7								Bleu ciel								
8									Brun							
9										Orange						
10											Gris clair					
11												Rose				
12													Vert			
13														Jaune		
14															Aqua	
15																Blanc

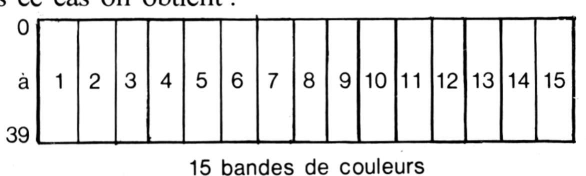
Visualisation de bandes verticales de couleurs

Le programme précédent peut être modifié pour donner des bandes verticales :



```
10      GR
20      FOR I = 0 TO 15
30      COLOR = I
40      FOR J = 0 TO 39
50      PLOT I, J
60      NEXT J
70      NEXT I
```

Dans ce cas on obtient :



15 bandes de couleurs

Reconnaissance de la couleur d'un point par programme

Il peut être nécessaire de vouloir connaître la couleur d'un point de coordonnées X, Y. Ceci est possible grâce à la fonction SCRN (X, Y), où X et Y peuvent avoir des valeurs entières de 0 à 39 (ou 47 si l'on utilise tout l'écran).

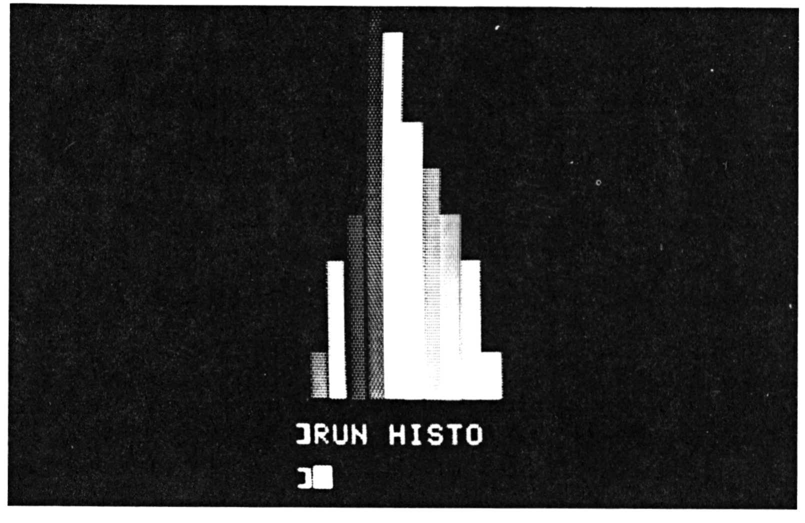
La valeur retournée par la fonction SCRN est alors un entier compris entre 0 et 15.

Application à la représentation d'un histogramme

Soit à représenter sous forme d'histogramme une population d'individus divisés en dix classes, de façon que la classe la plus grande utilise la hauteur maximum disponible sur l'écran.

On obtient le programme suivant :

```
10   DIM C(10),H(10)
15   REM LECTURE DES DONNEES DANS CHAQUE CLASSE
20   FOR I = 1 TO 10
30   READ C(I)
40   NEXT
50   DATA 10,30,40,100,80,60,50,40,30,10
55   REM RECHERCHE DE LA CLASSE MAXIMUM
60   FOR I = 1 TO 10
70   IF C(I) > C(I - 1) THEN IM = I : M = C(I)
80   NEXT I
90   HM = 39
95   REM CALCUL DE L'ECHELLE
100  EC = HM / M
105  REM CALCUL DE LA HAUTEUR DE CHAQUE CLASSE
110  FOR I = 1 TO 10
120  H(I) = INT (EC * C(I))
130  NEXT I
140  GR
145  REM VISUALISATION DE L'HISTOGRAMME
150  FOR I = 1 TO 10
160  COLOR = I + 1
170  FOR J = 39 TO 39 - H(I) STEP - 1
180  PLOT I,J
190  NEXT J,I
200  END
```



Déplacement d'un point sur l'écran

Jusqu'à présent nous avons vu des ordres de visualisation statiques. Or il peut être intéressant de faire bouger un point sur l'écran.

Pour cela, il faut changer l'emplacement du point et, si l'on veut visualiser la trajectoire du point, il n'y a rien d'autre à faire. Si, par contre, on ne veut pas visualiser la trajectoire, mais simplement l'emplacement du point en fonction du temps, on doit effacer l'ancienne position, ceci permet en particulier de faire de l'animation.

Exemples :

```
 10     GR
 20     COLOR = 10
 30     INPUT "ORIGINE";X,Y
 40     PLOT X,Y
 50     DX = 1
 60     DY = 1
 70     FOR I = 1 TO 10000
 80     X = X + DX
 90     Y = Y + DY
100     IF X > 39 THEN X = 0
110     IF Y > 39 THEN Y = 0
120     PLOT X,Y
130     NEXT I
200     END
```

Dans ce programme, le point se déplace en laissant une trace de sa trajectoire. Lorsque le point atteint un des bords de l'écran, il réapparaît de l'autre côté. Mais, une fois que la trajectoire est tracée, on ne voit plus le mouvement.

Si l'on veut seulement obtenir le mouvement du point, on remplacera les instructions 70 à 130 par le programme suivant :

```
 70     X1 = X + DX
 80     Y1 = Y + DY
 90     IF X1 > 39 THEN X1 = 0
100     IF Y1 > 39 THEN Y1 = 0
110     COLOR = 10
120     PLOT X1, Y1
130     COLOR = 0
140     PLOT X, Y
150     X = X1
160     Y = Y1
170     GO TO 70
180     END
```

Remarque. – D'une manière plus générale, on peut faire se déplacer un point de couleur *n* sur un fond de couleur différente.

Ainsi, le début du programme pourrait être :

```
INPUT "COULEUR FONDS" ; F
INPUT "COULEUR POINT" ; P
COLOR = F
FOR I = 0 TO 39
FOR J = 0 TO 39
PLOT I, J
NEXT J, I
```

L'instruction 110 deviendrait : COLOR = P
et l'instruction 130 : COLOR = F

```
  1   INPUT "COULEUR DU FOND";F
  2   INPUT "COULEUR DU POINT";P
  3   GR : COLOR = F
  4   FOR I = 0 TO 39
  5   FOR J = 0 TO 39
  6   PLOT I,J
  7   NEXT J,I
 10   COLOR = P
 20   HOME
 30   INPUT "ORIGINE";X,Y
 40   PLOT X,Y
 50   DX = 1
 60   DY = 1
 70   X1 = X + DX
 80   Y1 = Y + DX
 90   IF X1 > 39 THEN X1 = 0
100   IF Y1 > 39 THEN Y1 = 0
110   COLOR = P: PLOT X1,Y1
120   COLOR = F: PLOT X,Y
130   X = X1:Y = Y1
140   GOTO 70
200   END
```

Le Jeu de la vie

Le mathématicien CONWAY a proposé un jeu dit « Jeu de la Vie » en considérant des « cellules » susceptibles de se reproduire, de disparaître ou de survivre lorsqu'elles obéissent à certaines règles appelées « génétiques ». Ces cellules sont représentées par des éléments sur un damier dont la taille peut être arbitraire.

Chaque cellule est donc entourée par huit cases susceptibles de contenir d'autres cellules.

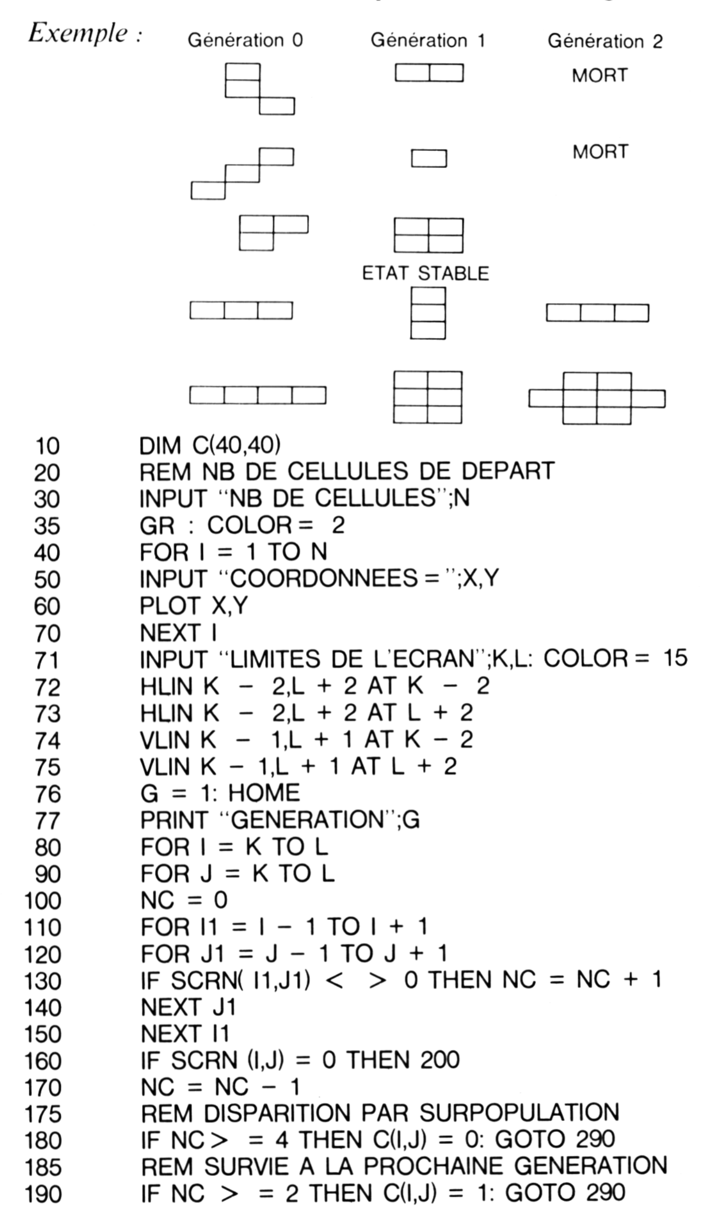
Les règles sont les suivantes :

1. La survie : Chaque cellule ayant deux ou trois cellules adjacentes survit jusqu'à la génération suivante.
2. La mort : Chaque cellule ayant quatre (ou plus) cellules adjacentes disparaît ou meurt par surpopulation. Chaque cellule n'ayant qu'une ou zéro cellule adjacente meurt d'isolement.

3. Naissance : Chaque emplacement adjacent à exactement trois cellules fait naître une nouvelle cellule pour la génération suivante.

Il est important de remarquer que toutes les naissances et les morts ont lieu en même temps au cours d'une génération.

Exemple :




```
10    DIM C(40,40)
20    REM NB DE CELLULES DE DEPART
30    INPUT "NB DE CELLULES";N
35    GR : COLOR= 2
40    FOR I = 1 TO N
50    INPUT "COORDONNEES = ";X,Y
60    PLOT X,Y
70    NEXT I
71    INPUT "LIMITES DE L'ECRAN";K,L: COLOR= 15
72    HLIN K - 2,L + 2 AT K - 2
73    HLIN K - 2,L + 2 AT L + 2
74    VLIN K - 1,L + 1 AT K - 2
75    VLIN K - 1,L + 1 AT L + 2
76    G = 1: HOME
77    PRINT "GENERATION";G
80    FOR I = K TO L
90    FOR J = K TO L
100   NC = 0
110   FOR I1 = I - 1 TO I + 1
120   FOR J1 = J - 1 TO J + 1
130   IF SCRN( I1,J1) < > 0 THEN NC = NC + 1
140   NEXT J1
150   NEXT I1
160   IF SCRN (I,J) = 0 THEN 200
170   NC = NC - 1
175   REM DISPARITION PAR SURPOPULATION
180   IF NC> = 4 THEN C(I,J) = 0: GOTO 290
185   REM SURVIE A LA PROCHAINE GENERATION
190   IF NC > = 2 THEN C(I,J) = 1: GOTO 290
```

```
194    REM MORT PAR ISOLEMENT
195    C(I,J) = 0
196    REM NAISSANCE
200    IF NC = 3 THEN C(I,J) = 1
290    NEXT J,I
300    REM VISUALISATION NOUVELLE GENERATION
310    FOR I = K TO L
320    FOR J = K TO L
330    IF C(I,J) = 0 THEN COLOR= 0: GOTO 350
340    COLOR= 2
350    PLOT I,J
360    NEXT J,I
370    G = G + 1
375    HOME
380    PRINT "GENERATION";G
390    GOTO 80
500    END
```

Exercices

1. *Modifier le programme de déplacement d'un point sur l'écran pour le faire « rebondir » sur les côtés de l'écran.*
2. *Ecrire un programme de marche aléatoire.*
   Un point est entouré de huit positions possibles notées de 1 à 8. La marche aléatoire consiste à tirer au hasard le « pas » suivant parmi ces huit positions.

1	*2*	*3*
4		*5*
6	*7*	*8*

3. *Même exercice en choisissant une couleur aléatoire à chaque pas.*

CORRIGES

N° 1

```
 1     INPUT "COULEUR DU FOND";F
 2     INPUT "COULEUR DU POINT";P
 3     GR : COLOR= F
 4     FOR I = 0 TO 39
 5     FOR J = 0 TO 39
 6     PLOT I,J
 7     NEXT J,I
10     COLOR= P
20     HOME
30     INPUT "ORIGINE";X,Y
40     PLOT X,Y
```

```
50    DX = 1
60    DY = 1
70    X1 = X + DX
80    Y1 = Y + DY
85    REM SI L'ON ARRIVE AUX BORDS REBONDIR
90    IF X1 = 0 OR X1 = 39 THEN DX = - DX
100   IF Y1 = 0 OR Y1 = 39 THEN DY = - DY
105   REM VISUALISER LE NOUVEAU POINT
110   COLOR = P: PLOT X1,Y1
115   REM EFFACER L'ANCIEN POINT
120   COLOR = F: PLOT X,Y
130   X = X1:Y = Y1
140   GOTO 70
200   END
```

N° 2

```
1     INPUT "COULEUR DU FOND";F
2     INPUT "COULEUR DU POINT";P
3     GR : COLOR = F
4     FOR I = 0 TO 39
5     FOR J = 0 TO 39
6     PLOT I,J
7     NEXT J,I
10    COLOR = P
20    HOME
30    INPUT "ORIGINE";X,Y
35    PLOT X,Y
40    REM INITIALISER LES TABLEAUX DX ET DY
41    DIM DX(8),DY(8)
42    FOR I = 1 TO 8
43    READ DX(I),DY(I)
44    NEXT I
45    DATA - 1, - 1,0, - 1,1, - 1, - 1,0,1,0, - 1,1,0,1,1,1
50    REM CHOIX DE L'INCREMENT ALEATOIRE
60    K = INT (8 * RND (1) + 1)
70    X1 = X + DX(K)
80    Y1 = Y + DY(K)
85    REM SI L'ON ARRIVE AUX BORDS PASSER DE L'AUTRE
      COTE
90    IF X1 < 0 THEN X1 = 39
95    IF X1 > 39 THEN X1 = 0
100   IF Y1 < 0 THEN Y1 = 39
101   IF Y1 > 39 THEN Y1 = 0
105   REM VISUALISER LE NOUVEAU POINT
110   COLOR = P: PLOT X1,Y1
115   REM EFFACER L'ANCIEN POINT
120   COLOR = F: PLOT X,Y
130   X = X1:Y = Y1
140   GOTO 60
200   END
```

N° 3

```
  1   INPUT "COULEUR DU FOND";F
  2   INPUT "COULEUR DU POINT";P
  3   GR : COLOR = F
  4   FOR I = 0 TO 39
  5   FOR J = 0 TO 39
  6   PLOT I,J
  7   NEXT J,I
 10   COLOR = P
 20   HOME
 30   INPUT "ORIGINE";X,Y
 35   PLOT X,Y
 40   REM INITIALISER LES TABLEAUX DX ET DY
 41   DIM DX(8),DY(8)
 42   FOR I = 1 TO 8
 43   READ DX(I), DY(I)
 44   NEXT I
 45   DATA -1,-1,0,-1,1,-1,-1,0,1,0,-1,1,0,1,1,1
 50   REM CHOIX DE L'INCREMENT ALEATOIRE
 60   K = INT (8 * RND (1) + 1)
 70   X1 = X + DX(K)
 80   Y1 = Y + DY(K)
 85   REM SI L'ON ARRIVE AUX BORDS PASSER DE L'AUTRE
      COTE
 90   IF X1 < 0 THEN X1 = 39
 95   IF X1 > 39 THEN X1 = 0
100   IF Y1 < 0 THEN Y1 = 39
101   IF Y1 > 39 THEN Y1 = 0
105   REM VISUALISER LE NOUVEAU POINT
106   P = INT (15 * RND (1) + 1)
110   COLOR = P: PLOT X1,Y1
115   REM EFFACER L'ANCIEN POINT
120   COLOR = F: PLOT X,Y
130   X = X1:Y = Y1
140   GOTO 60
200   END
```

Le tracé de segments de droite en graphique simple

L'on a vu que l'on pouvait tracer des segments de droite verticaux par programme.

De même, si l'on voulait tracer un segment de droite horizontale sur la douzième ligne, on écrirait :

```
10   GR
15   COLOR = 10
20   FOR J = 0 TO 39
30   PLOT 12, J
40   NEXT J
50   END
```

Sur la plupart des systèmes graphiques existent des instructions permettant de réaliser ces tracés à l'aide d'une seule instruction.

Tracé d'un segment horizontal

Ceci peut être réalisé par l'instruction HLIN (ligne horizontale).

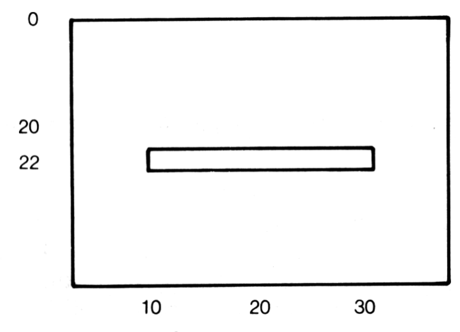
La syntaxe en est :

HLIN expression 1, expression 2 AT expression 3

- expression 1 représente l'abscisse de l'origine du segment (valeur entière de 0 à 39) ;
- expression 2 représente l'abscisse de la fin du segment (entre 0 et 39) ;
- expression 3 représente la valeur de l'ordonnée à laquelle doit être tracé le segment horizontal (compris entre 0 et 39 ou 0 et 47).

Exemple : HLIN 10, 30 AT 22

Tracer un segment de droite horizontale d'ordonnée 22 allant de l'abscisse 10 à l'abscisse 30.



Tracé d'un segment vertical

Une instruction similaire permet de tracer un segment vertical. La syntaxe en est :

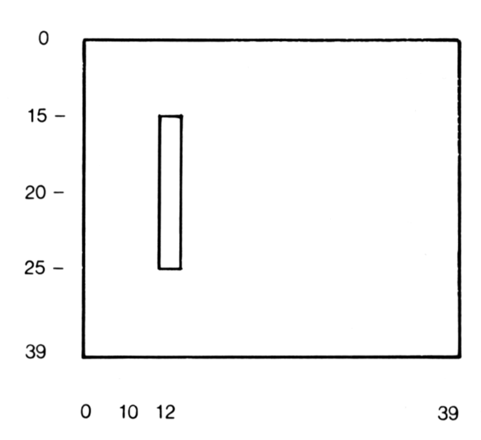
VLIN expression 1, expression 2 AT expression 3

- expression 1 et expression 2 représentent les ordonnées des extrémités du segment à tracer ;
- expression 3 représente l'abscisse du segment vertical à visualiser.

Exemple :

```
10    I = 15 : J = 25 : A = 12
20    GR : COLOR = 8
30    VLIN I, J AT A
40    END
```

Ce programme permet de visualiser un segment vertical d'abscisse 12 entre les ordonnées 15 et 25.



Tracé d'un segment de droite quelconque en pointillé :

Soit la droite d'équation $y = ax + b$.

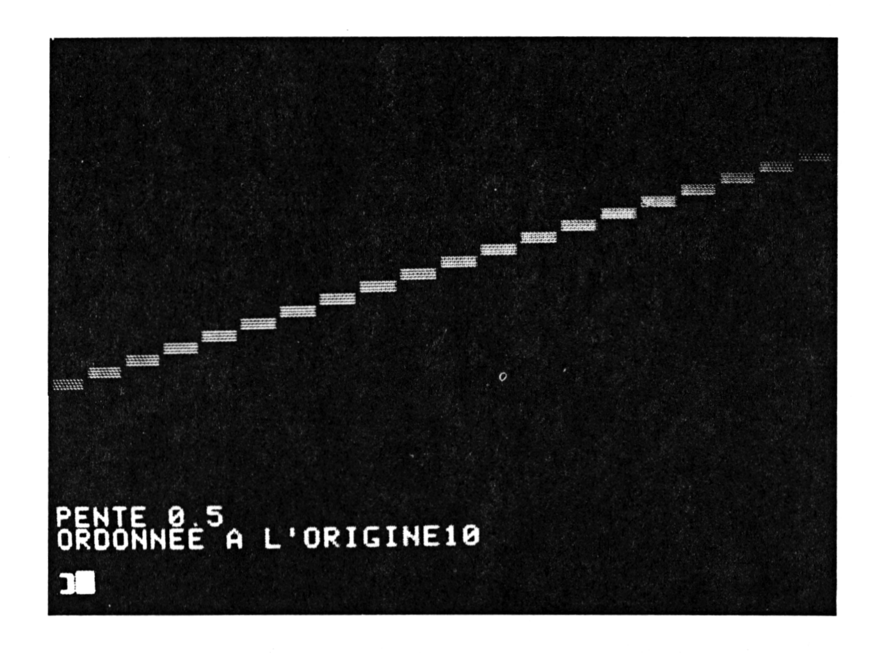
En mode graphique simple, il n'existe pas d'instruction permettant de tracer une droite quelconque, car la grille de points est trop grossière. On ne peut donc obtenir que des tracés très approchés d'une droite sous forme de pointillés. Le Programme est alors :

```
1       INPUT "PENTE";A
101     INPUT "ORDONNEE A L'ORIGINE";B
15      GR : COLOR = 5
20      FOR X = 0 TO 39
30      Y = 39  -  INT (A * X + B)
35      IF Y  < 0 OR Y > 39 THEN 50
40      PLOT X,Y
50      NEXT X
60      END
```

Remarque. – Il faut utiliser Y = 39 – ENT ($ax + b$), car l'axe des y est dans le sens contraire du sens habituel.

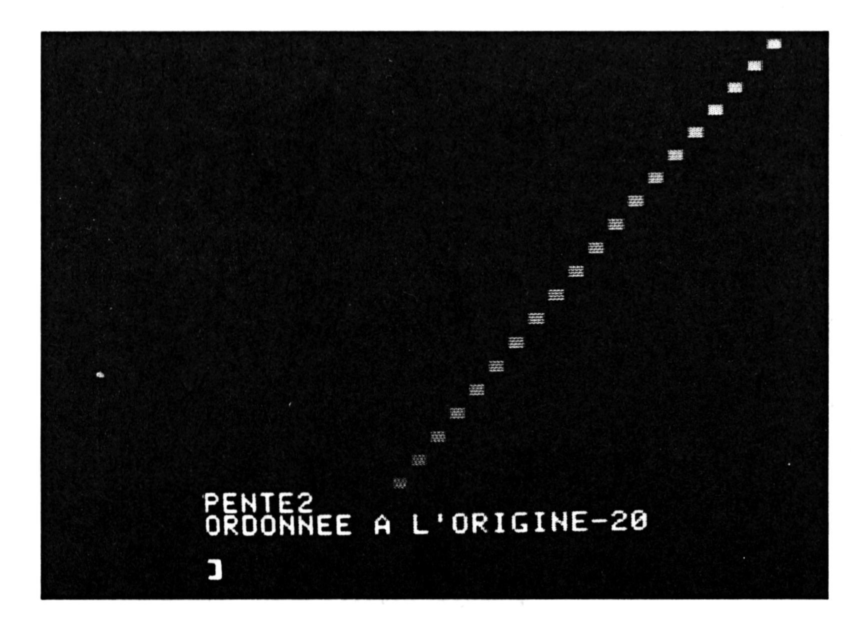
Exemple d'exécution :

PENTE ? 0,5, ®   Droite Y = X/2 + 10
ORDONNEE A L'ORIGINE ? 10 ®

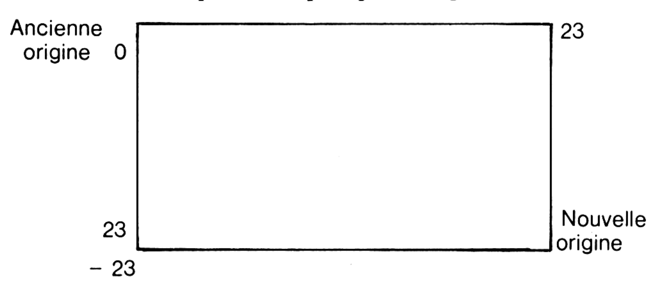


PENTE ? 2 ®
ORDONNEE A L'ORIGINE ? − 20 ®

Droite Y = 2 X − 20



Si l'on avait considéré une droite d'équation : Y = − X − 1, on n'aurait pas pu la tracer avec le programme précédent, car, pour obtenir des valeurs de Y positives, il aurait fallu donner à X des valeurs négatives. Dans ce cas il faut opérer un changement d'origine et considérer par exemple que l'origine est en bas à droite.



La droite tracée a alors comme équation :

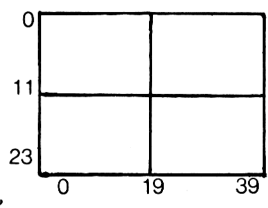
$$39 - Y = -(X - 39) - 1 = -X + 39 - 1 = -X + 38$$

d'ou Y = X + 1 avec la nouvelle origine

Le tracé de courbes en mode graphique simple

Là aussi, la taille de la grille ne permet d'avoir que des courbes approchées dans les meilleurs des cas.

Pour pouvoir être certain d'obtenir toutes les portions de courbes situées dans les quatre quadrants, on devra pouvoir opérer des changements d'axes. Ainsi, en considérant que le point central d'abscisse 19 et d'ordonnée 19 est l'origine des axes, on peut avoir une idée de l'allure de la courbe, puis indiquer un changement d'origine plus approprié.



Changement d'origine

Si la nouvelle origine a pour coordonnées x_0, y_0, on sait que la nouvelle équation d'une courbe $y = f(x)$ devient :

$$y - y_0 = f(x - x_0)$$

Comme le sens de y est inversé on a donc :

$$-y + y_0 = f(x - x_0)$$

Exemple :

- Si l'on prend $x_0 = 0$ et $y_0 = 39$ on obtient :

$$Y = 39 - f(x)$$

• Si l'on prend $x_0 = 20$ $y_0 = 20$, on a :

$$Y = 20 - f(x - 20)$$

Application

Soit à représenter la parabole $y = x^2$, on obtient :

```
10    DEF FN YP(X) = X ↑ 2
15    INPUT "ORIGINE";X0,Y0
20    INPUT "ECHELLE";E
30    GR : COLOR = 1
40    PLOT X0,Y0
50    REM VISUALISER LA COURBE
60    COLOR = 2
70    FOR X = 0 TO 39
80    A = X - X0
90    Y = INT (Y0 - FN YP(A) * E + 0.5)
100   IF Y < 0 OR Y > 39 THEN 120
110   PLOT X,Y
120   NEXT X
130   END
```

Remarque. – Le graphique simple est peu adapté aux tracés de courbes. Nous y reviendrons en utilisant le graphique haute résolution.

L'interaction avec un système graphique

Dans le cas où l'on dispose d'une zone au bas de l'écran, qui n'est pas utilisée pour le graphique, il est possible de rentrer et de sortir du texte dans cette "fenêtre". Cependant, cette fenêtre ne permet pas de pointer directement sur la partie graphique de l'écran. Les mécanismes d'interaction sur un système graphique sont divers : on peut citer le crayon lumineux, le manche à balai ("joy stick"), la souris ("mouse"), le crayon associé à une table à dessin..., etc.

Dans tous les cas, il faut pouvoir repérer les coordonnées d'un point (X, Y) grâce à un mécanisme contrôlé par l'homme. Le système le plus simple, qui est utilisé sur beaucoup de machines de jeux et sur le système "APPLE", est de disposer de manettes associées à des potentiomètres (résistances variables) qui donnent des valeurs X et Y. Les fonctions associées sont appelées PDL (du nom de "Paddle" = "pagaie" ou manette). Ainsi, sur un système graphique interactif, on peut disposer de deux manettes dont les positions définissent les coordonnées d'un point.

Ces fonctions sont appelées par programme sous la forme PDL (0) ou PDL (1) et retournent une valeur entière allant de 0 à 256.

Par exemple : X = PDL (0) et Y = PDL (1).

Il peut exister jusqu'à quatre manettes.

La syntaxe de cette instruction est :

PDL (expression entière)

mais, de façon pratique, la valeur de l'expression doit être comprise entre 0 et 3.

Dans le cas où la valeur de l'expression est comprise entre 4 et 255, le résultat de la fonction PDL est imprévisible.

Application au positionnement d'un point sur l'écran

Du fait que la valeur retournée par la fonction PDL est comprise entre 0 et 255, et que les coordonnées d'un point vont de 0 à 39, il faut diviser la valeur obtenue par un facteur

$$\frac{255}{39} = 6.$$

Si l'on divise donc la fonction PDL par 6, on obtient donc des valeurs comprises entre

$$0 \text{ et } \frac{255}{6} = 42.$$

Il suffit donc d'écrire par exemple :

```
X = PDL(0)/6
IF     X > 39          THEN      X = 39
Y = PDL(1)/6
IF     Y > 39          THEN      Y = 39
```

Ainsi est-on assuré que l'on peut balayer tous les points de l'écran.

On peut également écrire :

```
X = PDL(0)*39/255
Y = PDL(1)*39/255
```

Exemple :

Tracé d'une esquisse sous contrôle des manettes :

```
10     INPUT "COULEUR" ; C
20     GR : COLOR = C
30     X = PDL(0) * 39/255
40     Y = PDL(1) * 39/255
50     IF X = X0 AND Y = Y0 THEN 30
60     PLOT X, Y
70     X0 = X
80     Y0 = Y
90     GO TO 30
100    END
```

Exercices

1. *Ecrire un programme de jeux de « Mastermind ». Le programme choisit quatre pions parmi six couleurs : BLEU, ROUGE, VERT, JAUNE, ORANGE, MAUVE. Il peut y avoir plusieurs pions de même couleur. Le joueur indique son choix, à l'aide d'une suite de lettres correspondant à la première lettre de la couleur :*

   (B, R, V, J, O, M).

2. *Ecrire un programme de jeu de carré chinois ; chaque joueur a une couleur :*

1	2	3
4	5	6
7	8	9

   Le joueur place les points de sa couleur pour essayer de faire une ligne de trois points tandis que l'autre joueur l'en empêche. Les cases du carré sont numérotées comme ci-dessus.

3. *Les tours de Hanoï :*
   Initialement existe une tour composée de disques ou anneaux de diamètres décroissants.

   Le but est de transporter ces anneaux pour constituer une autre tour en ayant seulement le droit de déposer des anneaux de plus petit diamètre sur une autre pile.

   A un instant donné il ne peut y avoir plus de trois tours ou piles de disques.

LE GRAPHIQUE DE HAUTE RESOLUTION

On a vu que l'utilisation du graphique de faible résolution était quelque peu limitée lorsque l'on voulait tracer des courbes. Le graphique de haute résolution utilise une grille de points beaucoup plus petits et permet donc d'obtenir des tracés beaucoup plus précis.

Nous avons vu que le graphique de haute résolution sur le système APPLE utilisait une grille de points de 280 sur 160, avec une fenêtre de texte.

Le passage en mode graphique avec fenêtre se fait à l'aide de la commande :

HGR

tandis que le passage en mode graphique pour tout l'écran se fait à l'aide de la commande.

HGR2

La couleur

Le jeu de couleurs est plus limité et est spécifié par la commande :

HCOLOR = Expression entière.

Il existe huit possibilités qui sont codées par les entiers de 0 à 7 :

0 = noir	4 = noir
1 = bleu	5 dépend du téléviseur
2 = vert	6 dépend du téléviseur
3 = blanc	7 = blanc

Comme on le voit il y a deux blancs et deux noirs, tandis que les autres couleurs peuvent varier d'un téléviseur à un autre et suivant la technique utilisée pour l'interface avec le téléviseur.

Visualisation d'un point

C'est l'instruction HPLOT qui existe de façon similaire à l'instruction graphique PLOT.

Le format en est alors :

HPLOT expression 1, expression 2

expression 1 et expression 2 représentent les coordonnées (X et Y) des points à visualiser.

Exemple :

Soit à visualiser les fonctions SINUS et COSINUS. On peut écrire le programme :

```
 10    HGR
 20    FOR I = 0 TO 20 STEP 0. 1
 25    REM ECHELLE A 50 SUR Y
 30    K = INT (50 * (SIN(I) + 1))
 40    J = INT (I * 10)
 50    L = INT (50 * (COS (I) + 1))
 60    HCOLOR = 2
 70    HPLOT J, K
 80    HCOLOR = 1
 90    HPLOT J, L
100    NEXT I
```

La génération de segments de droite (vecteurs).

Le graphique haute résolution suppose également la possibilité d'engendrer des segments de droite entre deux points de coordonnées (x_1, y_1) et (x_2, y_2).

Ceci pourrait bien sûr se faire en calculant l'équation de la droite qui est donnée par :

$$\frac{x - x_1}{y - y_1} = \frac{x_2 - x_1}{y_2 - y_1}$$

En pratique, les systèmes graphiques de haute résolution fournissent des facilités de génération de vecteurs de ce type.

Ainsi, sur le système APPLE l'instruction HPLOT peut être utilisée sous la forme :

TRACER JUSQU'A expression 1, expression 2

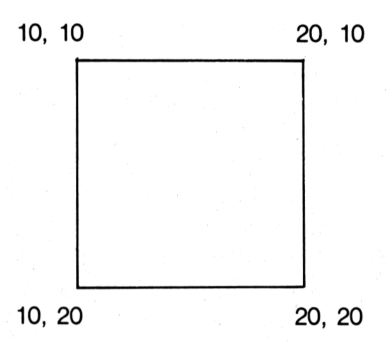
qui indique que l'on veut tracer un vecteur depuis le point précédemment visualisé jusqu'au point de coordonnées (x, y) spécifié par les expressions 1 et 2 qui suivent le mot TO (JUSQU'A).

La forme la plus générale de l'instruction HPLOT permet de demander le tracé de plusieurs vecteurs qui se suivent :

HPLOT exp 1, exp 2 TO exp 3, exp 4...
TO exp k, exp $k + 1$

Exemple :

Soit à tracer le carré dont les sommets sont donnés ci-dessous :



On écrit :

```
10  HGR
20  HGOLOR = 2
30  HPLOT 10, 10, TO 10, 20 TO 20, 20 TO 20, 10 TO 10, 10
```

Remarques. – Les expressions correspondant aux abscisses doivent avoir des valeurs comprises entre 0 et 279, tandis que celles correspondant aux ordonnées doivent être comprises entre 0 et 159 ou 191, suivant la taille de la grille utilisée.

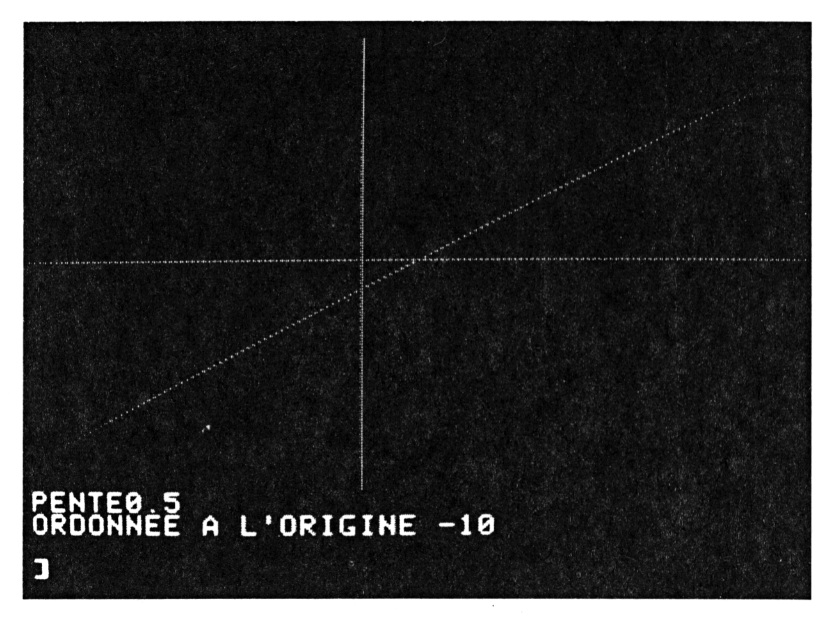
Tracé de droites

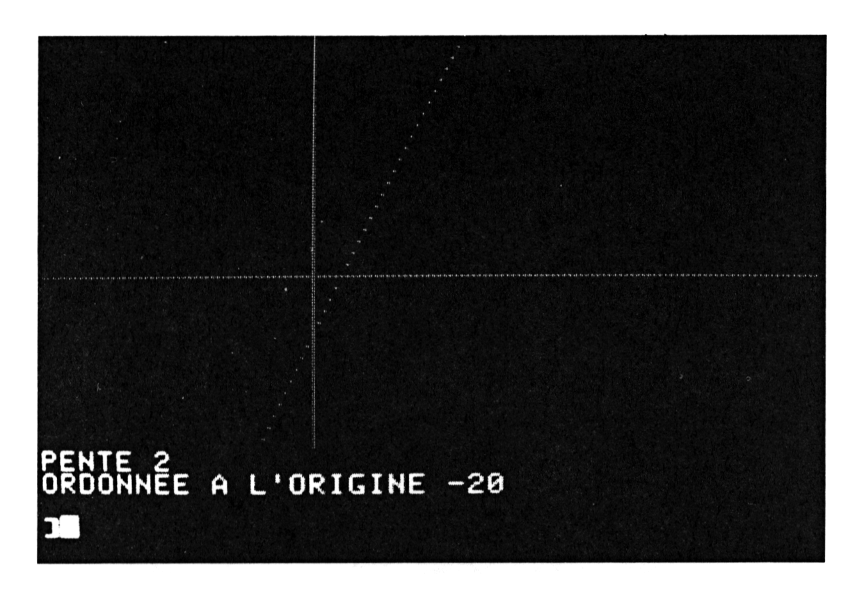
Si l'on reprend le problème de tracé de droite $y = ax + b$, cette fois, on obtient des tracés beaucoup plus rigoureux.

En reprenant le programme déjà écrit pour le graphique de basse résolution on a :

```
1    REM TRACE DE DROITE
10   INPUT "ORIGINE";X0,Y0
20   HGR : HCOLOR = 2
30   REMTRACE DES AXES
40   HPLOT 0,Y0 TO 279,Y0
50   HPLOT X0,0 TO X0,159
60   INPUT "PENTE";A
70   INPUT "ORDONNEE A L'ORIGINE" ;B
80   FOR X = 0 TO 279
85   X1 = X - X0
90   Y = Y0 - INT (A * X1 + B)
100  IF Y < 0 OR Y > 159 THEN 120
110  HPLOT X,Y
120  NEXT X
200  END
```

Exemples :





Tracé de polygones réguliers

Un polygone régulier peut être inscrit dans un cercle ; les coordonnées de ses sommets peuvent donc être obtenues par des fonctions trigonométriques. Si le rayon du cercle enveloppant le polygone est R, les coordonnées des sommets sont R cos A et R sin A, où A est l'angle sous lequel est vu un côté du polygone à partir du centre. A = $2\,\pi/n$, où n est le nombre de côtés.

Exemple :

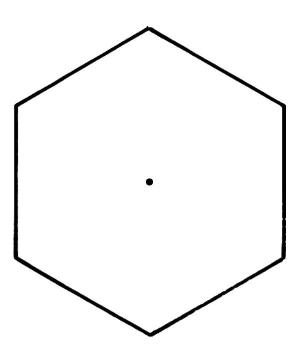
Pour tracer le polygone, il suffit donc de joindre tous les points ainsi obtenus.

Le programme obtenu est alors :

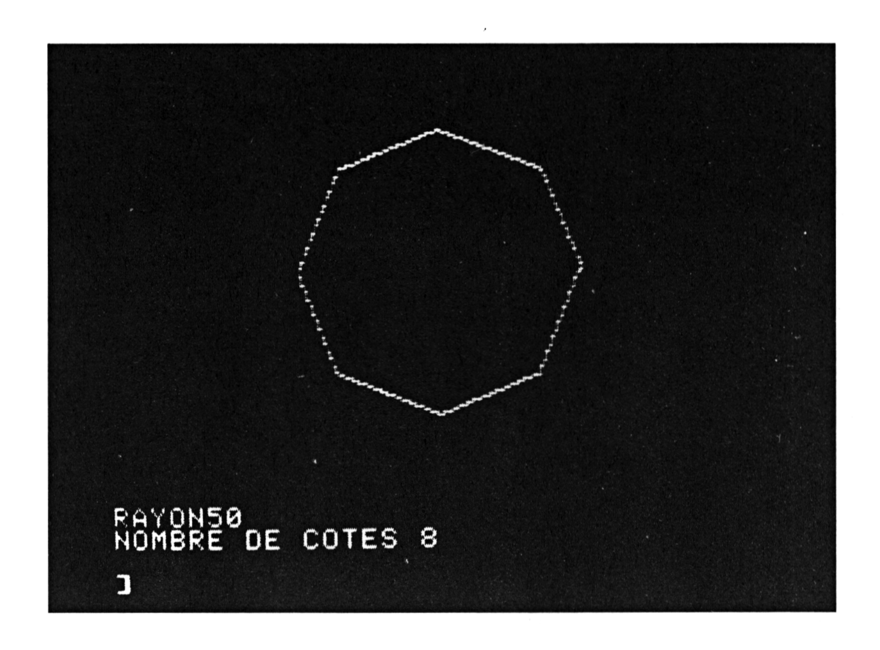
```
1      REM TRACE DE POLYGONE REGULIER
10     INPUT "ORIGINE";X0,Y0
20     INPUT "RAYON";R
30     INPUT "NOMBRE DE COTES";N
40     HGR : HCOLOR = 3
50     PI = 3.14
60     A = 2 * PI / N
70     HPLOT X0 + R,Y0
75     FOR I = 1 TO N
80     X = X0 + R * COS (A * I)
90     Y = Y0 - R * SIN (A * I)
110    HPLOT TO X,Y
120    NEXT I
200    END
```

Exécution :

ORIGINE	100,	80	®
RAYON		20	®
NOMBRE DE COTES		6	®

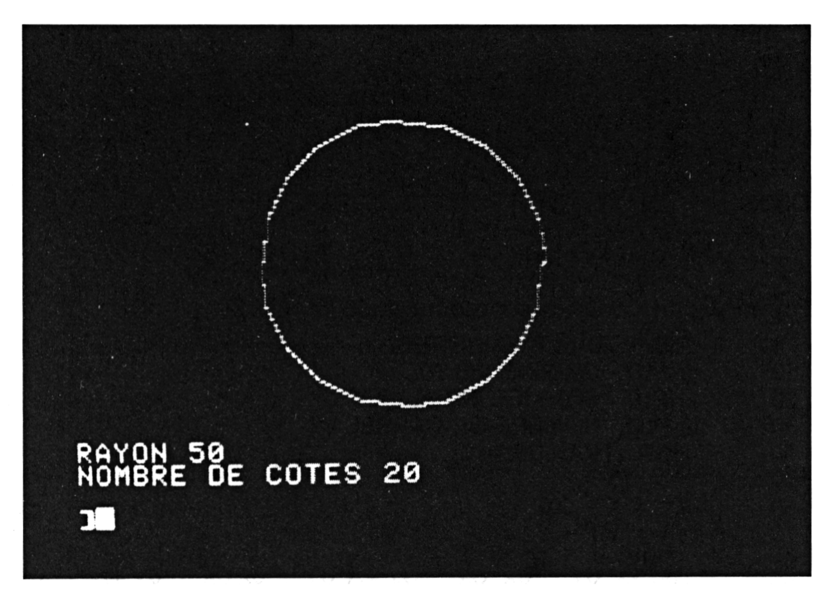


Tracé d'un hexagone



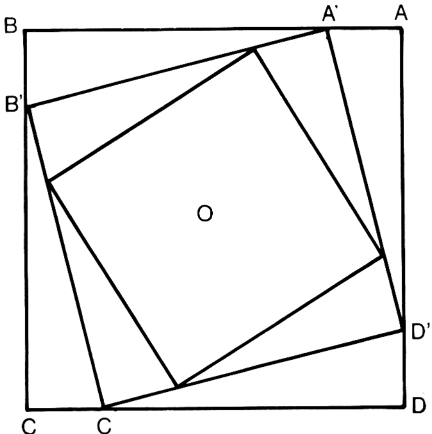
Application au tracé d'un cercle

Si l'on utilise le programme suivant avec N = 100, on obtient une figure très proche d'un cercle.



Exercice :

Ecrire un programme permettant de tracer des carrés emboîtés tels que :



Pour cela, il faut et il suffit que AA' = BB' = CC' = DD'. On a d'autre part

$$\begin{aligned} OA' &= OA + AA' = OB' = OB + BB' \\ &= OC' = OC + CC' = OD' = OD + DD' \end{aligned}$$

D'une manière générale, si les coordonnées de A sont (x, y), celles de B sont $(y, -x)$, celles de C = $(-x, -y)$ et celles de D = $(-y, x)$.

Comme :

$$AA' = \frac{AB}{K} \qquad AB = OB - OA = (y - x, \ -x - y)$$

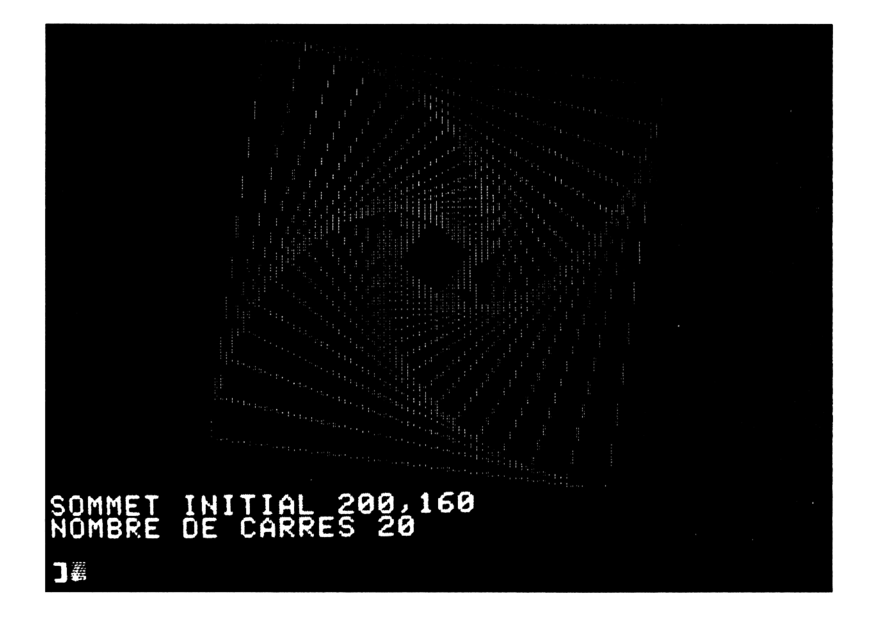
$$AA' = \frac{y - x}{K}, \ - \frac{x + y}{K}$$

Donc :

$$\vec{OA'} = \vec{OA} + \vec{AA'} = x + \frac{y - x}{K}; y - \frac{y + x}{K}$$

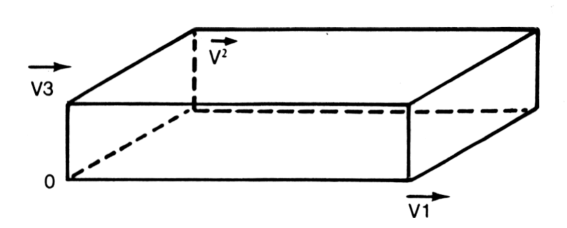
On en déduit donc les coordonnées de B', C', D'.

```
1     REM CARRES EMBOITES
10    INPUT "ORIGINE";X0,Y0
20    INPUT "SOMMET INITIAL";X,Y
30    INPUT "NOMBRE DE CARRES";N
40    HGR : HCOLOR= 2
50    K = 10
60    X1 = X - X0
70    Y1 = Y0 - Y
80    FOR I = 1 TO N
90    HPLOT X1 + X0, Y0 - Y1 TO Y1 + X0,Y0 + X1 TO - X1
      + X0,Y0 + Y1 TO - Y1 + X0,Y0 - X1
95    HPLOT TO X1 + X0,Y0 - Y1
100   X1 = X1 - (X1 - Y1) / K
120   Y1 = Y1 - (X1 + Y1) / K
130   NEXT I
```



Figures dans l'espace

Soit à tracer un parallélépipède :

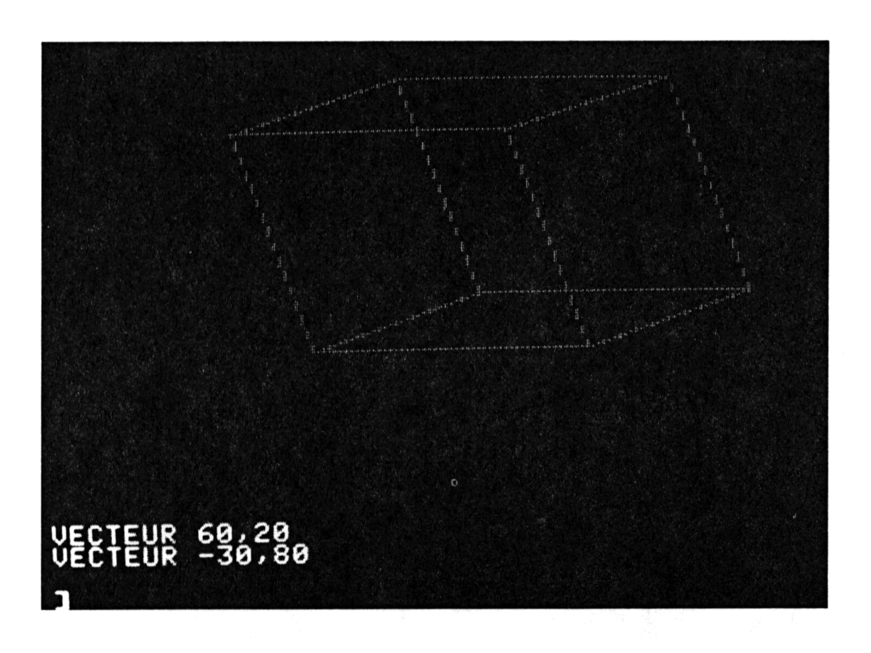


Les sommets d'un tel objet sont donnés par des combinaisons des trois vecteurs $\vec{V}1$, $\vec{V}2$, $\vec{V}3$.

Chaque sommet est tel que :

$$OM = b_1\vec{V}_1 + b_2\vec{V}_2 + b_3\vec{V}_3$$

avec $bi = 0$ ou 1.



Les sommets peuvent donc être représentés par les triplets de valeur *bi*. Ces triplets sont représentés sur le schéma ci-dessus. On peut d'autre part vérifier que si l'on part du sommet (0, 0, 0), il y a donc trois directions possibles pour tracer un nouveau sommet, puis à partir de (1, 0, 0) ou 0, 0, 1) ou (0, 1, 0) il y a deux directions possibles dans chacun des cas.

De manière générale, pour un sommet (b_1, b_2, b_3) il y a autant d'arêtes qui partent qu'il y a de $bi = 0$. D'autre part, la direction des vecteurs qui partent de ce sommet correspond au vecteur V_i.

Ainsi, par exemple :

- du sommet 0, 0, 1 partent des arêtes équipolentes à V_1 et V_2,
- du sommet 1, 1, 0 part une arête équipolente à V_3.

On rappelle qu'un vecteur $\vec{V}$ peut être caractérisé par ses composantes VX et VY sur les axes des x et des y.

$$\vec{V} = VX \times \vec{i} + VY \times \vec{j}$$

où $\vec{i}$ et $\vec{j}$ sont les vecteurs unitaires.

La somme des deux vecteurs est obtenue en ajoutant les composantes respectives des deux vecteurs.

Remarque. – Cette méthode peut être généralisée pour le tracé d'un hyperparallélépipède à partir d'un nombre de n vecteurs.

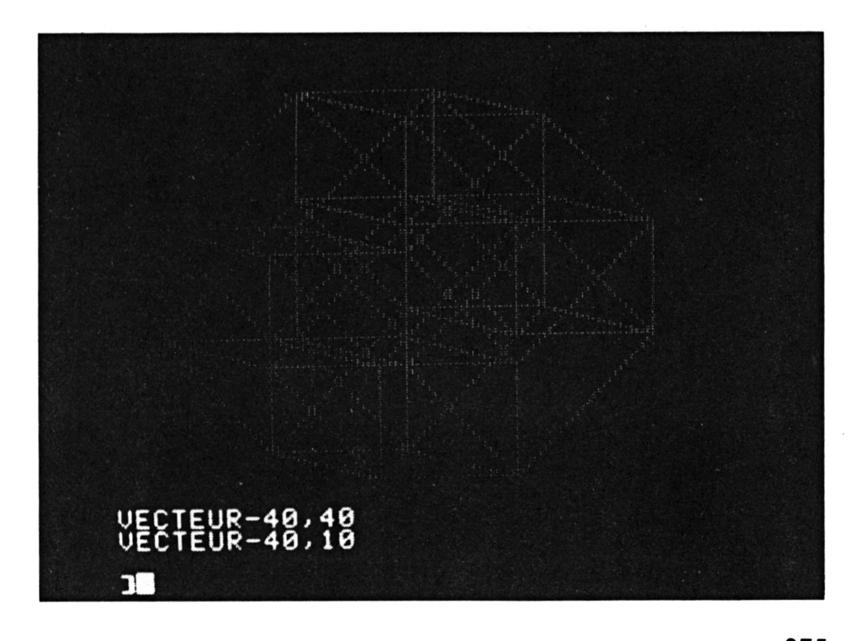
Il suffit pour cela de modifier le programme précédent en introduisant une variable N au lieu de trois. Le nombre de sommets est alors de 2^N.

On obtient alors le programme suivant pour le tracé d'un hyperparallélépipède.

```
1     REM TRACE D'HYPERPARALLELIPIPEDE
5     DIM X(100),Y(100)
10    INPUT "ORIGINE";X0,Y0
15    INPUT "DIMENSIONS";K
20    HGR : HCOLOR= 2
30    HPLOT X0,Y0
40    FOR I = 0 TO K - 1
50    INPUT "VECTEUR";VX(I), VY(I)
60    B(I) = 0
70    NEXT I
75    REM CONSIDERER LES 2↑N SOMMETS
80    FOR I = 0 TO 2 ↑ K - 1
```

```
84    REM CONVERSION EN BINAIRE
85    N = I
90    FOR J = K - 1 TO 0 STEP - 1
100   B(J) = INT (N / 2 ↑ J)
110   N = N - B(J) * 2 ↑ J
120   NEXT J
124   REM CALCUL COORDONNEES DU SOMMET I
125   X(I) = X0:Y(I) = Y0
130   FOR J = 0 TO K - 1
150   IF B(J) = 0 THEN 180
160   X(I) = X(I) + VX(J)
17Q   Y(I) = Y(I) - VY(J)
180   NEXT J
185   REM TRACE DES VECTEURS ISSUS DU SOMMET I
190   FOR J = 0 TO K - 1
200   IF B(J) < > 0 THEN 240
210   XV(J) = X(I) + VX(J)
220   YV(J) = Y(I) - VY(J)
230   HPLOT X(I),Y(I) TO XV(J),YV(J)
240   NEXT J
250   NEXT I
300   END
```

Programme traçant un hyperparallélépipède



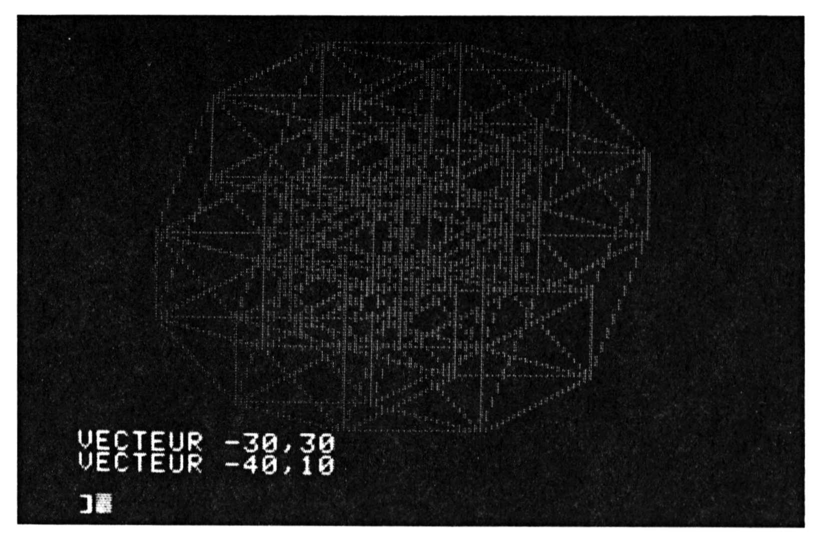


Figure obtenue pour un hyperparallélépipède

Tracés de courbes

En graphique haute résolution, il est possible de tracer n'importe quelle courbe algébrique avec une bonne approximation.

Comme on l'a vu pour le graphique de basse résolution, le problème principal est celui de l'échelle.

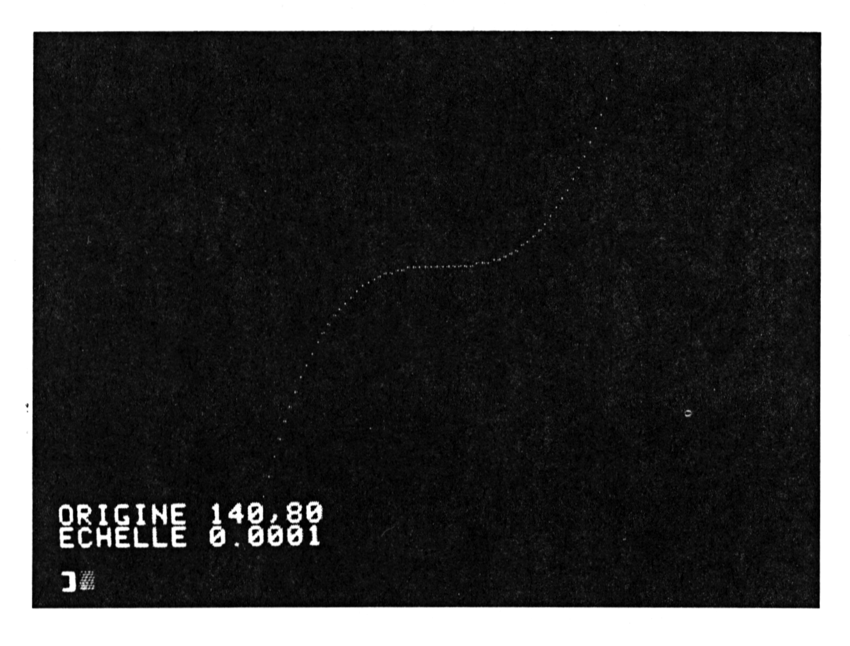
Lorsque l'on veut visualiser une courbe, on commence donc par utiliser une échelle de 1 puis, en fonction des résultats obtenus, on peut la modifier pour obtenir une courbe plus adaptée à la taille de l'écran.

Application au tracé de fonctions polynomiales.

```
  1    REM TRACE DE FONCTIONS POLYNOMIALES
 10    INPUT "ORIGINE";X0,Y0
 20    DEF FN FP(X) = X↑5 - 3*X↑4 + 2*X↑3 - X + 1
 30    INPUT "ECHELLE";E
 40    HGR : HCOLOR = 2
 42    REM TRACE DES AXES
 43    HPLOT 0,Y0 TO 279,Y0
 44    HPLOT X0,0 TO X0,159
 50    FOR X = 0 TO 279
 60    A = X - X0
 70    Y = INT (Y0 - FN FP(A) * E + 0.5)
 80    IF Y < 0 OR Y > 159 THEN 100
 90    HPLOT X,Y
100    NEXT X
200    END
```

Exécution :

ORIGINE 140,80 ®
ECHELLE 0.0001 ®

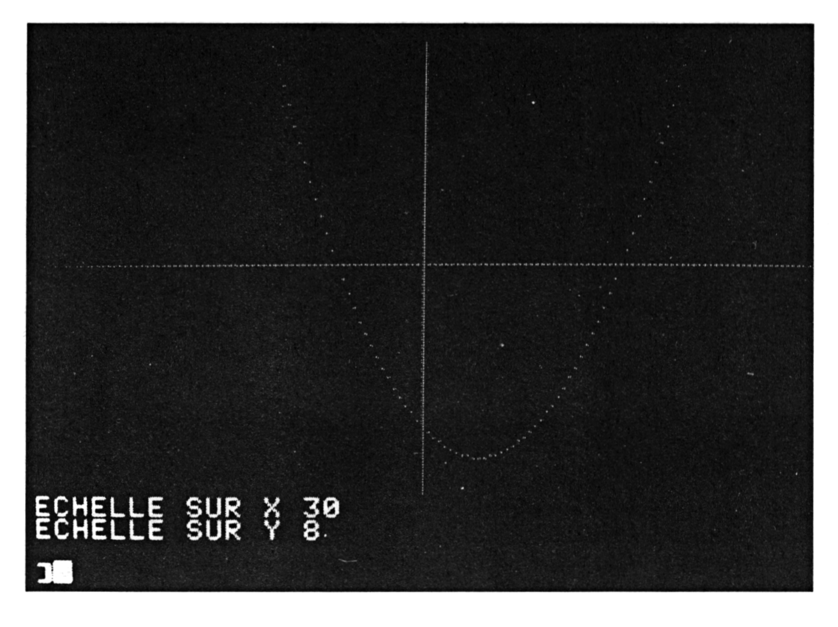




Ainsi on peut également introduire un facteur d'échelle sur l'axe des Y, ce qui donne :

```
1     REM TRACE DE FONCTIONS POLYNOMIALES
10    INPUT "ORIGINE";X0,Y0
20    DEF FN FP(X) = 3 * X ↑ 2 - 4 * X - 7
30    INPUT "ECHELLE SUR X";K
35    INPUT "ECHELLE SUR Y";E
40    HGR : HCOLOR= 2
42    REM TRACE DES AXES
43    HPLOT 0,Y0 TO 279,Y0
44    HPLOT X0,0 TO X0,159
50    FOR X = 0 TO 279
60    A = X - X0
65    A = A / K
70    Y = INT (Y0 - FN FP(A) * E + 0.5)
80    IF Y < 0 OR Y > 159 THEN 100
90    X NEXT X
200   END
```

Exemple :



Problème des fonctions qui prennent des valeurs infinies pour des valeurs finies

Le programme précédent ne peut être utilisé pour tracer le graphe de la fonction $y = \frac{1}{x}$

En effet, pour $x = 0$ on a $y = \infty$, et dans ce cas l'on aura une erreur de division par zéro.

Le branchement sur erreur

Pour remédier à ce problème, il existe souvent une possibilité qui permet de se brancher à un traitement spécifique lorsqu'il y a une erreur d'exécution. Cette instruction est un branchement conditionnel sur une erreur. La syntaxe en est :

ONERR GO TO *n*
(SUR ERREUR ALLER A n)

n est le numéro de l'instruction où l'on doit se brancher si une erreur se produit.

Le type d'erreur est spécifié par un code qui peut être lu grâce à une instruction PEEK (voir ci-dessous) à une adresse spécifique.

Cette instruction doit être exécutée avant l'apparition de l'erreur. Ainsi, si l'on sait qu'une fonction prend des valeurs infinies à cause de divisions par zéro, il suffit d'insérer une telle instruction.

La continuation après erreur

Si l'on veut reprendre l'instruction sur laquelle l'erreur s'est produite on utilisera l'instruction RESUME (REPRISE)

Application

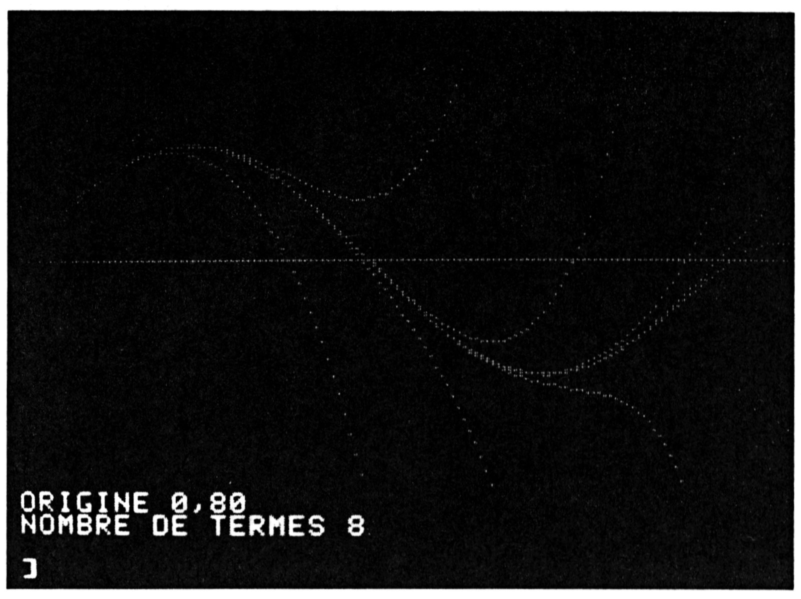
Ceci peut être utilisé pour le tracé de fonctions rationnelles de la forme :

$$y = \frac{ax^2 + bx + c}{a'x^2 + b'x + c'}$$

La modification du programme de tracé de courbes déjà vu est laissée au lecteur à titre d'exercice.

Développements limités de fonctions

On sait que certaines fonctions de variables réelles peuvent être approchées par des séries polynomiales.



Par exemple :

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots\ldots + \frac{x^n}{n!} + \ldots\ldots$$

$$\frac{1}{1+x} = 1 - x + x^2 - x^3 + x^4 + \ldots\ldots \text{si } 0 < x < 1$$

$$\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots\ldots + (-1)^{n-1} \frac{x^{2n+1}}{(2n+1)!}$$

$$\text{Log}\,(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} + \ldots\ldots \text{pour } x < 1$$

Il est intéressant de vérifier graphiquement que ces polynômes donnent des courbes de plus en plus rapprochées de la fonction qu'ils approximent.

Ainsi, par exemple, pour sin x on a le programme suivant :

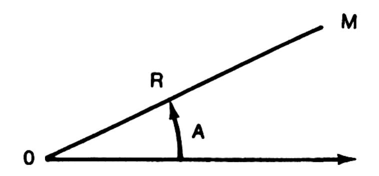
```
1     REM DEVELOPPEMENTS LIMITES
10    INPUT "ORIGINE";X0,Y0
20    INPUT "NOMBRE DE TERMES";N
40    HGR : HCOLOR= 2
42    REM TRACE DES AXES
43    HPLOT 0,Y0 TO 279,Y0
44    HPLOT X0,0 TO X0,159
50    FOR X = 0 TO 7 STEP 0.025
60    Y = O:F = 1
```

```
70     HPLOT X - X0,Y0 - Y
80     K = 1
90     FOR I = 0 TO N
100    DL = K * X ↑ (2 * I + 1) / F
110    K = - K
120    F = F * 2 * (I + 1) * (2 * I + 3)
125    A = X * 40
130    Y = Y + DL * 40
135    IF Y0 - Y < 0 OR Y0 - Y > 159 THEN 150
140    HPLOT A - X0,Y0 - Y
150    NEXT I
160    NEXT X
200    END
```

Les tracés en coordonnées polaires

Certaines courbes s'expriment beaucoup plus simplement en utilisant des coordonnées polaires, où l'on donne la distance R = OM par rapport à une origine en fonction de l'angle A avec l'axe d'origine.

R est appelé le rayon vecteur et A l'angle polaire.



Une courbe en coordonnées polaires est donc exprimée par une fonction du type :

$$R = f(A)$$

Exemples :

- Un cercle centré sur 0 est exprimé en coordonnées polaires par R = Constante.
- R = K (1 − cos A) est une courbe appelée cardioïde.
- R = KA est une courbe appelée spirale d'Archimède.
- R = K sin 2 A est une rosace à quatre branches.

Le passage de coordonnées polaires aux coordonnées cartésiennes est très simple puisque le point M a des coordonnées x et y telles que :

$$x = R \cos A$$
$$y = R \sin A$$

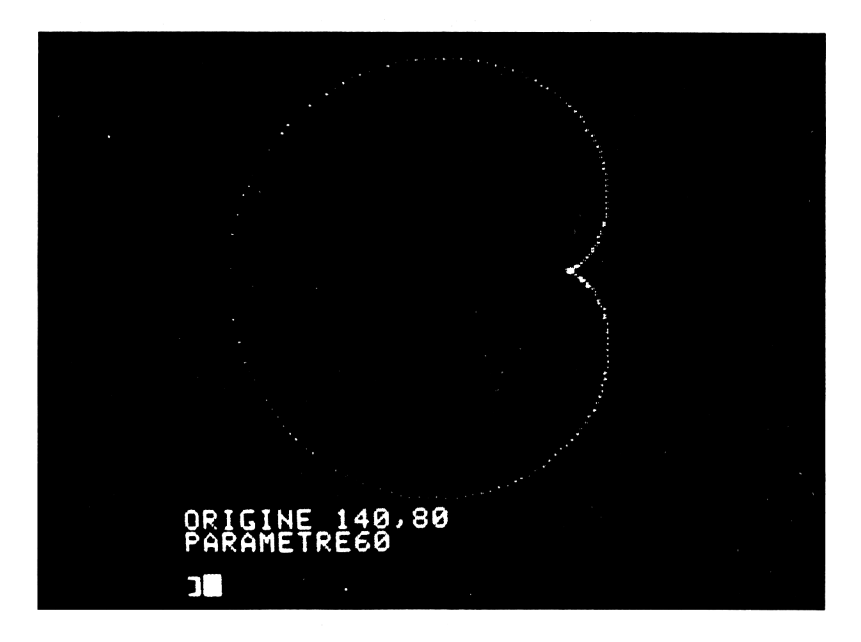
Ainsi devient-il possible de se donner des courbes en coordonnées polaires et de les tracer en utilisant l'instruction HPLOT.

Soit à tracer la courbe R = K (1 − cos A); on obtient le programme suivant :

```
1    REM TRACE DE CARDIOIDE
10   INPUT "ORIGINE";X0,Y0
20   INPUT "PARAMETRE";K
25   HGR : HCOLOR= 3
26   PI = 3.14
30   FOR A = 0 TO 2 * PI STEP PI / 20
40   X = K * (1 - COS (A)) * COS (A)
50   Y = K * (1 - COS (A)) * SIN (A)
60   HPLOT X + X0,H0 - Y
70   NEXT A
80   END
```

On obtient une courbe dont le graphe est donné ci-dessous :

Tracé d'une cardioïde



Tracé de rosaces et de « fleurs »

On peut montrer que la famille de courbes dont l'équation polaire est

$$R = K + \cos n A$$

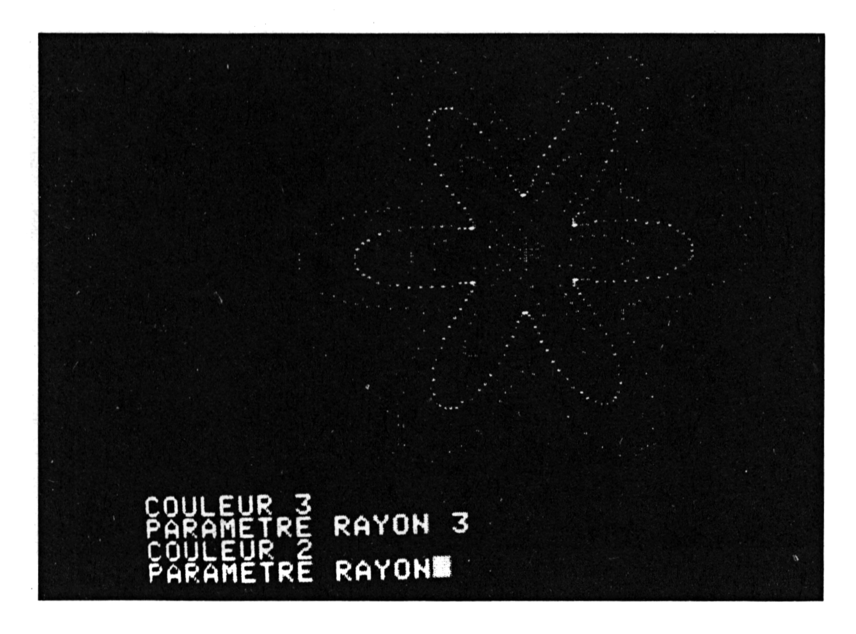
représente les projections de la trajectoire d'un point se déplaçant régulièrement sur un tore.

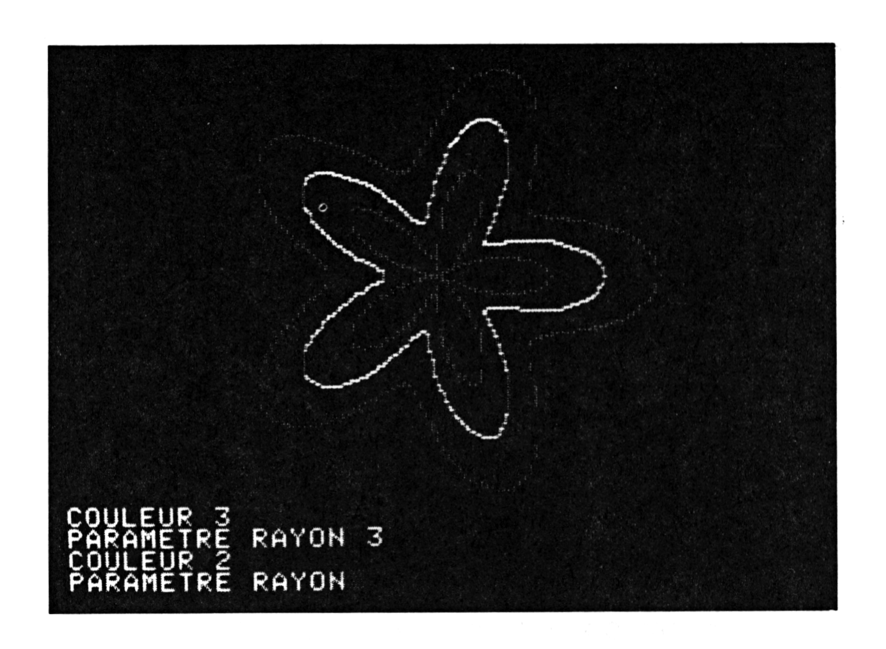
Les courbes correspondantes sont des courbes ayant des formes de rosaces ou de fleurs.

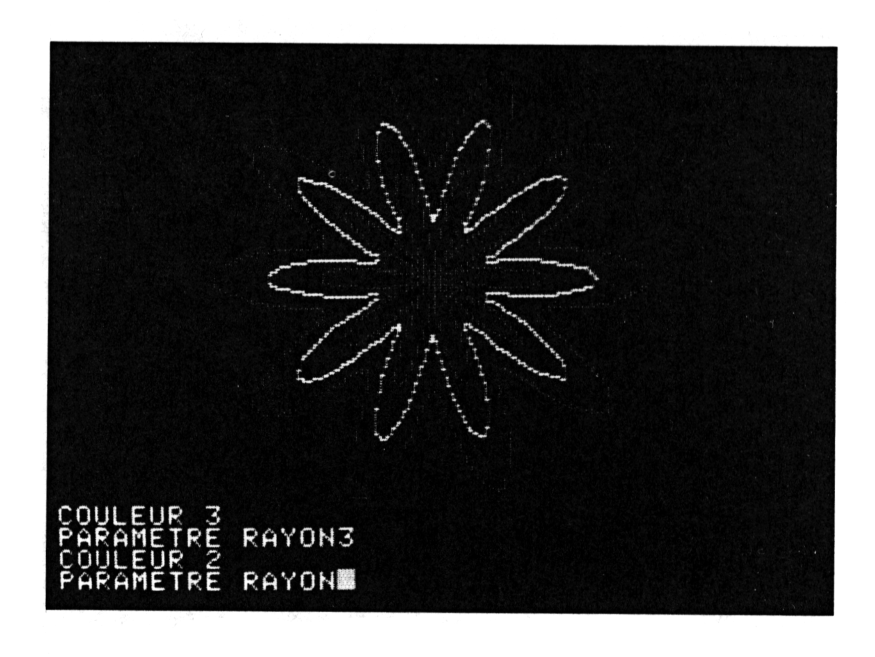
Le programme correspondant est :

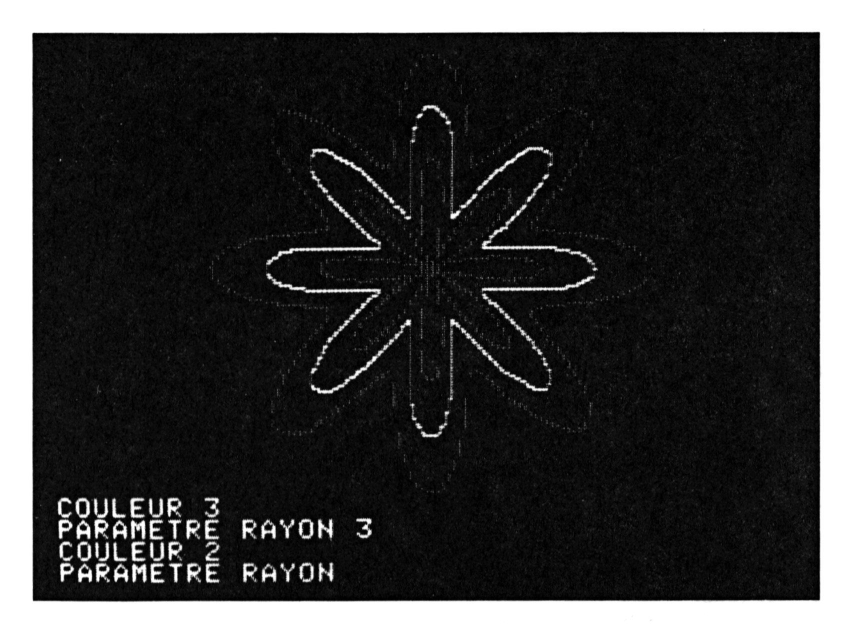
```
  1   REM TRACE DE ROSACES
 10   INPUT "ORIGINE";X0;Y0
 20   INPUT "PARAMETRE ANGULAIRE";N
 26   PI = 3.14
 30   HGR
 40   INPUT "PARAMETRE RAYON";K
 50   INPUT "COULEUR";C
 55   HCOLOR= C
 56   HPLOT X0 + 20 * K + 20,Y0
 60   FOR A = 0 TO 2 * PI STEP PI / 100
 70   X = (K + COS (N * A)) * COS (A)
 80   Y = (K + COS (N * A)) * SIN (A)
 85   X = X * 20
 86   Y = Y * 20
 90   HPLOT TO X + X0,Y0 - Y
100   NEXT A
110   GOTO 40
200   END
```

Exemples :




COULEUR 3
PARAMETRE RAYON 3
COULEUR 2
PARAMETRE RAYON


COULEUR 3
PARAMETRE RAYON3
COULEUR 2
PARAMETRE RAYON



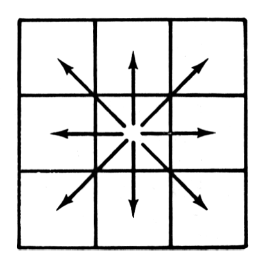
La définition de formes graphiques.

Dans les exemples précédents nous avons traité des courbes qui peuvent s'exprimer par des équations algébriques. Nous avons vu également comment traiter des images que l'on peut construire de façon algorithmique.

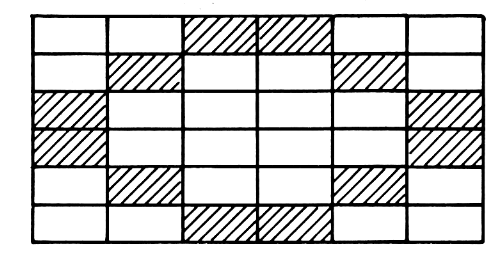
Cependant, si l'on veut créer des formes définies point par point, puis travailler sur ces formes de manière algorithmique, il nous faut définir des mécanismes permettant de les définir et de les traiter.

Quel que soit le système utilisé, une forme est un ensemble de points, et, si l'on dispose bien d'une instruction permettant de visualiser point par point, il est intéressant de pouvoir définir une forme par un processus continu.

Sur une grille de points, le déplacement peut se faire dans huit directions données ci-dessous.



Soit à définir la forme suivante :



Si l'on part du haut dans le sens des aiguilles d'une montre on a donc les mouvements suivants :

→↘↘↓↙↙←↖↖↑↗↗

On peut attribuer un codage à ces mouvements par exemple :

→ D (droite)
← G (gauche)
↑ H (haut)
↓ B (bas)
↗ DH (droite haut)
↘ DB (droite bas)
↗ GB (gauche bas)
↘ GH (gauche haut)

La séquence précédente peut alors être codée par la suite :

D, DB, DB, B, GB, GB, G, GH, GH, H, DH, DH

Programme de création et visualisation d'une forme

En utilisant le système de codage précédent, on se propose de créer et de visualiser une forme. On rajoute deux lettres A pour annuler le mouvement précédent et T pour terminer. Le programme suivant permet de créer et de visualiser une forme :

```
 10    DIM F$(100)
 20    INPUT "ORIGINE";X0,Y0
 30    HGR : HCOLOR = 2
 40    HPLOT X0,Y0
 50    X = X0:Y = Y0
 55    FOR I = 1 TO 1000
 60    INPUT F$(I)
 70    IF F$(I) = "A" THEN HCOLOR = 0: HPLOT X,Y: GOTO 60
 80    IF F$(I) = "T" THEN 220
 90    HCOLOR = 2
100    IF F$(I) = "D" THEN X = X + 1: GOTO 200
110    IF F$(I) = "G" THEN X = X - 1: GOTO 200
120    IF F$(I) = "H" THEN Y = Y - 1: GOTO 200
```

```
130   IF F$(I) = "B" THEN Y = Y + 1: GOTO 200
140   IF F$(I) = "GB" THEN X = X - 1:Y = Y + 1: GOTO
      200
150   IF F$(I) = "DB" THEN X = X + 1:Y = Y + 1: GOTO
      200
160   IF F$(I) = "DH" THEN X = X + 1:Y = Y - 1: GOTO
      200
170   IF F$(I) = "GH" THEN X = X - 1:Y = Y - 1: GOTO
      200
175   GOTO 60
176   REM
180   REM VISUALISER LE POINT X,Y
190   REM
200   HCOLOR= 2: HPLOT X,Y
210   NEXT I
220   END
```

Codage interne de la forme

A partir du tableau F$, il est possible de créer un tableau de codes internes, il suffit en effet de trois bits pour représenter l'ensemble des huit déplacements possibles.

Sur le système APPLE, le système de codage des déplacements est sensiblement différent. En effet, il suppose que l'on se limite à des déplacements à angle droit. Si le point est visualisé, on l'indique à l'aide d'une flèche pointée.

Le système de codage utilisé est alors :

↑ 000	•↑ 100
→ 001	•→ 101
↓ 010	•↓ 110
← 011	•← 111

Il est facile de passer du système de codage précédent à celui-ci, en effet, les déplacements obliques sont l'équivalent de deux déplacements orthogonaux :

DH = ↗ = →↑

DB = ↘ = →↓

GH = ↖ = ←↑

GB = ↙ = ←↓

Il est ainsi possible d'engendrer une forme codée dans une table que l'on appelle table de forme.

Sur le système APPLE on regroupe deux ou trois codes dans un octet, suivant que l'on a des déplacements avec ou sans visualisation. La table de formes peut en contenir plusieurs et est disposée à un endroit précis de la mémoire.

Les instructions de manipulation de forme.

Il existe une instruction qui permet de tracer une forme à partir d'une table en mémoire.

TRACER expression 1 A expression 2, expression 3
(DRAW) (AT)

L'expression 1 spécifie le numéro d'ordre de la forme dans la table.

Les expressions 2 et 3 spécifient les coordonnées de l'origine.

Il existe une deuxième version qui permet de faire un tracé de couleur complémentaire à chaque point visualisé. C'est l'instruction C TRACER OU XDRAW. Les couleurs complémentaires sont Noir et Blanc ou Bleu et Vert. L'intérêt de cette instruction est que deux instructions CTRACER (XDRAW) successives permettent d'effacer une forme sans modifier l'environnement.

Les autres instructions sont les instructions de changement d'échelle et de rotation de la forme.

La mise à l'échelle 0 ou le changement d'échelle se font par l'instruction :

SCALE (ECHELLE) = expression

L'expression est une valeur entière précisant l'échelle à utiliser (entre 0 et 255).

La rotation d'une forme se fait à l'aide de l'instruction :

ROT = expression.

La valeur de l'expression peut varier de 0 à 255, mais pour une échelle de 1 seules les valeurs, 0, 16, 32, 48, sont reconnues et permettent de faire des rotations de 90°, 180° ou 270° dans le sens des aiguilles d'une montre. Pour une échelle de 2 il y a huit rotations possibles..., etc.

Lorsqu'une valeur n'est pas reconnue, on utilise la valeur correcte la plus proche :

Exemple :

```
10      HGR
20      H COLOR = 2
30      FOR I = 1 TO 64
40      ROT = I
50      SCALE = I
60      DRAW 1 AT 100, 80
70      NEXT I
```

Ce programme permet de tracer une forme à partir du point de coordonnées 100, 80, en faisant varier l'échelle et en faisant tourner la forme.

CONCLUSION

Ce chapitre est sans doute le plus original dans un ouvrage d'introduction au BASIC.

En effet, le graphique est bien souvent considéré, soit comme un luxe, soit comme un "gadget" inutile à l'apprentissage de la programmation.

Dans le premier cas, on pense à des périphériques très coûteux, et, dans le deuxième cas, on pense à des procédés semi-graphiques utilisés pour les jeux vidéos.

Le développement de systèmes graphiques sur téléviseur standard a permis de montrer que ce n'est plus un luxe et que le graphique pouvait être utilisé pour des programmes sérieux et des jeux !

En particulier, le graphique permet de présenter des résultats d'une manière beaucoup plus parlante que des tableaux de chiffres.

Enfin, le graphique possède une qualité essentielle qui ne peut être remplacée par aucun autre moyen de sortie et qui est la représentation du temps sous forme de mouvement. Ceci peut être mis à profit non seulement pour des jeux, mais également pour la représentation dynamique de résultats, la simulation de processus dynamiques, l'animation et la représentation de formes artistiques ou géométriques.

CHAPITRE VII

LES INSTRUCTIONS SPECIFIQUES A CERTAINS SYSTEMES BASIC

BASIC est sans doute le langage qui possède le plus de variantes dans les extensions qu'offrent les différents systèmes. Dans ce chapitre, nous ne prétendons donc pas lister toutes ces variantes. Nous nous limiterons aux instructions qui se retrouvent sur plusieurs systèmes et qui ont une utilité certaine.

LES INSTRUCTIONS "SYSTEME"

Nous entendons par là certaines instructions qui permettent d'accéder à la mémoire de la machine ou de spécifier le branchement à un sous-programme en langage machine.

Lecture d'une mémoire

C'est l'instruction :

PEEK (adresse mémoire)

L'adresse mémoire peut être spécifiée par une expression entière qui varie de 0 à la taille maximum de la mémoire de la machine.

L'intérêt de cette instruction est de permettre de lire certaines positions mémoire ayant des fonctions spécifiques sur une machine donnée (entrée-sortie).

La valeur lue est retournée en décimal (0 à 255) et doit être mémorisée dans une variable :

X = PEEK (AD)

APPLICATIONS :

Création d'un son

Sur certaines machines disposant d'un speaker, celui-ci peut être mis en route à l'aide de cette instruction.

Si S est l'adresse mémoire associée à ce speaker, on peut produire un son en écrivant l'instruction :

SON = PEEK (S)

On peut également créer un son plus long en écrivant :

BRUIT = PEEK(S) – PEEK(S) + PEEK(S) – PEEK(S)

Position du curseur

Il est en général possible d'avoir la position horizontale et verticale du curseur à l'aide d'instructions :

CH = PEEK(H)
CV = PEEK(V)

H et V sont des adresses (36 et 37 sur APPLE par exemple).

Lecture d'un caractère au clavier

Si C est l'adresse associée au clavier l'instruction

X = PEEK(C)

permet d'obtenir le code ASCII de la touche qui a été appuyée (avec le bit 7 positionné). En particulier, si $X > 127$, on sait qu'une touche a été appuyée.

Ecriture dans une mémoire

C'est l'instruction inverse de la précédente. Sa syntaxe en est :

POKE adresse prévue, valeur

Cette instruction sert à mettre une valeur dans la mémoire dont l'adresse est spécifiée.

Attention ! Si cette instruction est utilisée à tort et à travers, elle peut détruire des parties de programmes ou des données.

Il faut donc être prudent quand on l'utilise.

Exemples :

Pour placer le curseur à une position donnée, on peut utiliser des instructions POKE aux adresses déjà définies ci-dessus.

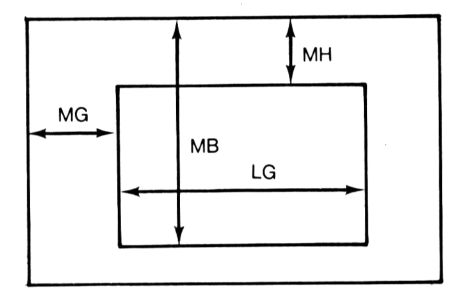
POKE H, CH
POKE V, CV

De même, il est possible de changer la fenêtre de texte sur l'écran en définissant la largeur de la fenêtre (1 à 40) et une marge à gauche ainsi qu'en haut et en bas.

Ainsi, les instructions suivantes définissent :

POKE F, LG La largeur de la fenêtre
POKE G, MG La marge gauche
POKE H, MH la marge en haut.
POKE B, MB la marge en bas

(G = 32, F = 33, H = 34, B = 35 sur le système APPLE)



Exemple :

```
10      LG = 20 : MH = 10 : MB = 30 : MG = 10
20      G = 32 : F = 33 : H = 34 : B = 35
30      POKE F, LG
40      POKE G, MB
50      POKE B, MB
60      POKE H, MH
```

D'autres instructions POKE permettent de modifier des paramètres du système graphique, des contrôles de jeux, etc.

Remarque. – Les instructions PEEK et POKE existent sous des formes similaires sur la plupart des micro-ordinateurs individuels. Par contre, sur les plus gros systèmes, elles sont absentes, car dangereuses.

Définition de la mémoire utilisable par un programme BASIC

Il existe des instructions ou commandes permettant de fixer la borne inférieure et supérieure de la mémoire utilisable dans un programme BASIC

Ce sont les instructions :

HIMEM : adresse (mémoire la plus haute)
LOMEM : adresse (mémoire la plus basse)

Les adresses peuvent être spécifiées par des expressions entières qui correspondent à des adresses existantes. Si le système ne possède que 16 K, cette valeur sera comprise entre zéro et 16 384 s'il possède 64 K on pourra aller jusqu'à 65 535 (1 K = 2^{10} = 1 024).

On suppose que la valeur définie par LOMEM est inférieure à celle définie pour HIMEM.

Si ces instructions ne sont pas spécifiées le système utilise des bornes qui correspondent à toute la mémoire disponible pour l'utilisateur. L'effet de ces instructions est annulé par une opération de remise à zéro (RESET ou CONTROLE B).

L'instruction d'attente

Cette instruction permet de réaliser un délai programmé ou une pause. Cette instruction permet de s'arrêter dans un programme jusqu'à ce qu'une certaine condition soit vérifiée.

La syntaxe de cette instruction est :

WAIT adresse, expression 1, expression 2
(ATTENDRE)

Le premier paramètre est l'adresse d'une position mémoire qui sera testée pendant l'attente. Cette adresse peut être une expression à condition que cela corresponde à une adresse valide.

Les expressions 1 et 2 représentent des variables entières (de 0 à 255). L'expression 2 est facultative. Si la deuxième expression est absente, l'instruction réalise une opération logique ET entre le contenu de l'adresse et la valeur binaire de l'expression (bit à bit).

Tant que le résultat est nul, l'attente continue, tandis que si le résultat de l'opération ET est différent de 0 le délai d'attente s'achève.

Exemple :

WAIT AD, 3

Le nombre 3 est égal à 00000011 en binaire ; cette instruction provoquera donc un délai jusqu'à ce qu'au moins un des deux bits de droite soit présent dans la mémoire d'adresse AD. Si l'expression 2 est présente, il y aura d'abord une opération OU exclusif entre le contenu de la mémoire et la valeur de l'expression 2. L'opération OU exclusif est telle que si deux bits sont identiques, le résultat est nul, sinon le résultat est 1.

Ensuite, le résultat obtenu fera l'objet d'une opération ET avec la valeur de l'expression 1.

Ceci permet de tester une condition telle que un bit de la position mémoire soit nul.

Exemples :

- WAIT AD, 1, 1 spécifie une pause jusqu'à ce que le bit de droite de AD soit à 0 ;
- WAIT AD, 3, 2 spécifie une pause jusqu'à ce que le bit de droite de AD soit égal à 1 ou bien jusqu'à ce que le deuxième bit ait une valeur égale à 0.

L'appel à un sous-programme en langage machine

Cette instruction est utile lorsque l'on veut faire appel à un sous-programme écrit directement en langage machine. La syntaxe de l'instruction est :

CALL adresse
(APPEL)

L'adresse doit être un entier correspondant à une adresse existante sur la machine (si le système fait 64 K on peut aller jusqu'à 65 535). Cette adresse peut être représentée par une expression évaluée sous forme d'une valeur entière.

L'intérêt de cette instruction est de permettre de faire référence et d'exécuter des programmes écrits en langage machine ou en assembleur.

Certains programmes sont fournis par le constructeur, notamment pour l'effacement de l'écran, d'une ligne..., etc.

D'autre part, cela permet à l'utilisateur d'écrire des programmes plus spécifiques et de les appeler à partir d'un programme BASIC.

Le passage de paramètres

L'instruction précédente permet de faire un saut à un sou-programme assembleur, mais bien souvent il est nécessaire d'y associer des paramètres ou valeurs définies dans le programme appelant.

Ceci est possible grâce à une instruction du type (système APPLE)

USR (expression)

L'expression définit un paramètre qui peut être entier ou flottant (réel) et qui sera transmis au sous-programme assembleur par l'intermédiaire d'un accumulateur flottant qui se trouve en général dans la première page de la mémoire (adresses 0 à 255). Le détail de l'utilisation de cette instruction est donnée dans les brochures des constructeurs.

LES INSTRUCTIONS D'EDITION

Lorsque le système BASIC dispose d'une unité de visualisation (télévision ou moniteur TV) il est en général possible de définir des instructions d'édition spécifiques.

Mouvement du curseur

Le plus souvent, il existe des instructions permettant de positionner le curseur à un endroit précis de l'écran.

Ainsi, HTAB et VTAB permettent de positionner le curseur horizontalement et verticalement sur l'écran. La syntaxe en est :

HTAB X
VTAB Y

où X et Y sont des valeurs entières. Par exemple, si l'écran dispose de 24 lignes de 40 caractères, on doit avoir :

$$1 \leqslant X \leqslant 24 \text{ et } 1 \leqslant Y \leqslant 40$$

Effacement de l'écran

Il existe en général une instruction permettant d'effacer l'écran et de positionner le curseur en haut à gauche.

Cette instruction est une commande telle que : HOME (Retour « chez soi »). Dans certains cas, le même effet peut être obtenu en imprimant un caractère spécial. Ainsi, sur le CBM (PET) :

PRINT " ♡ "

aura le même effet.

Lecture de la position du curseur

Cette instruction permet de connaître la position du curseur sur la ligne courante. C'est l'instruction POS (expression) qui évalue la position du curseur par rapport à la marge gauche. La première position correspond à la valeur 0. L'expression peut être quelconque, mais n'a pas de signification pour POS.

Si l'on veut conserver cette position, on devra utiliser une instruction du type :

X = POS (0)

Exemple :

```
PRINT TAB (15)
X = POS(0)            Donne X = 14
PRINT X
```

Impression d'espaces

Bien qu'il soit possible d'imprimer des caractères blancs pour faire des éditions, il est préférable de disposer d'une instruction spécifique pour imprimer des espaces. Ainsi, l'instruction SPC (X) (SPACE = ESPACE) permet d'imprimer X espaces.

Alors que l'instruction TAB part du début de la ligne, l'instruction SPC permet d'imprimer X espaces à partir de la position courante du curseur. Si l'on dépasse le bord droit de la ligne on continue sur la ligne suivante.

La syntaxe en est :

SPC (expression)

L'expression est convertie en un entier de 0 à 255. SPC (0) n'a aucun effet (0 espace).

Il est cependant possible de réaliser l'impression d'un nombre quelconque d'espaces en juxtaposant plusieurs instructions SPC.

Exemple : PRINT SPC (255), SPC (100) permet d'imprimer 355 espaces.

Les commandes vidéo ou télévision

Sur les systèmes utilisant des appareils de TV standard en sortie, il existe des commandes telles ques :

FLASH

qui permettent de faire clignoter l'écran (passage de blanc sur noir à noir sur blanc).

La commande INVERSE permet de passer sans clignoter d'un mode où le texte est visualisé en blanc sur noir à un mode où il est visualisé en noir sur blanc. Le passage au mode blanc sur noir se fait alors en spécifiant la commande « NORMAL ».

La vitesse de défilement d'un texte ou d'une impression peut être spécifiée par une commande VITESSE (SPEED).

SPEED = expression

L'expression peut prendre des valeurs de 0 à 255.

La disponibilité de mémoire (FRE)

Sur la plupart des systèmes existe une commande ou instruction permettant de connaître l'espace mémoire disponible à l'utilisateur en dehors de la place occupée par son programme.

C'est l'instruction FRE (expr). En pratique, l'expression peut avoir la valeur 0. Le résultat est un nombre entier positif ou négatif. S'il est négatif, c'est une valeur supérieure à 32 767, que l'on obtient en rajoutant 65 536 (2^{16}).

Exemple : X = FRE (0) si X > 0 X est le nombre de mots.
X < 0 on rajoute 65 536

L'instruction d'impression avec format (PRINT... USING)

Cette instruction n'est pas toujours disponible sur les plus petits systèmes, mais elle est très souvent disponible sur des systèmes professionnels.

Elle est spécifiée par une clause de format introduite par le mot clé USING (UTILISANT).

Cette clause permet de contrôler le format d'édition en spécifiant une chaîne de caractères qui a une signification bien précise et donne la forme de l'édition que l'on veut obtenir. Dans cette suite de caractères, on indique la position des signes, des nombres, du point décimal, des virgules et des caractères de remplissage.

La syntaxe de cette instruction peut également varier légèrement d'un système à l'autre.

Dans la plupart des cas, elle fait référence à un numéro de format, mais on peut également spécifier le format dans une instruction PRINT

Exemples :

PRINT USING "# # # . # #", K permet d'imprimer K sous la forme d'un nombre de trois chiffres (ou blancs) avant le point, suivi de deux décimales.

On peut également avoir une forme du type

PRINT USING F$; X, Y, Z

associé à une instruction de format définie par :

F$ = "LES RESULTATS X, Y, Z SONT : $$# #.#, $$# #.#,
$$# #.#"

Enfin, on peut aussi écrire :

PRINT USING 40, A, B

Cette fois, le format est référencé par l'instruction 40, qui donne le format :

40 % A = # # #.# B = # # .#

Le jeu de caractères disponibles pour définir le format est également variable. Cependant les caractères les plus souvent utilisés sont :

#	:	Indique un chiffre, un blanc ou un signe −.
D	:	Imprime des chiffres 0 si le chiffre correspondant n'existe pas dans le nombre imprimé.
.	:	Le point indique le point (virgule) décimal.
,	:	La virgule indique l'impression effective d'une virgule.
$	:	Indique l'impression d'un caractère $ au début de la zone associée à un nombre. Si l'on spécifie $$ le $ sera accolé au nombre.
+	:	Indique l'impression d'un signe (+ ou −).
−	:	Imprime un signe − si le nombre est négatif ou un blanc si le nombre est positif.
*	:	Imprime un caractère * s'il n'y a pas de chiffre à cette place.

Remarque. − Cette liste de caractères peut légèrement varier d'un système à l'autre. Il est donc important de consulter les brochures du système utilisé pour l'utilisation de cette instruction.

L'avantage de cette instruction est de faciliter l'édition des résultats.

LA MISE AU POINT DES PROGRAMMES

En BASIC interprété, la mise au point de programmes est relativement aisée, quel que soit le système, puisque l'exécution d'un programme est interactive.

Bien souvent, il n'y a pas besoin de programmes spécifiques d'aide à la mise au point (programme de « debugging »).

En effet, s'il y a des erreurs de syntaxe, elles sont détectées à l'interprétation de l'instruction correspondante. S'il s'agit d'une erreur d'exécution, le message d'erreur indique l'instruction en cause.

Il faut noter cependant que, lorsque l'on ne comprend pas la cause d'une erreur, à l'exécution, il peut être nécessaire d'insérer des ordres de sorties (PRINT) de résultats intermédiaires juste avant l'instruction provoquant l'erreur. Ces instructions seront ensuite éliminées lorsque l'erreur aura été détectée.

Sur certains systèmes existent des possibilités de mise au point grâce à des commandes de type « TRACE » qui permettent d'obtenir l'impression ou la visualisation des instructions en cours d'exécution du programme.

Il peut également y avoir des possibilités de placer des points d'arrêts dans un programme (« break-points ») permettant de vérifier ce qui s'est passé, puis de reprendre l'exécution éventuellement en pas à pas, c'est-à-dire instruction par instruction.

Ces facilités ne sont fournies que sur les systèmes professionnels et sont en particulier nécessaires pour les versions de BASIC qui sont compilées. En effet, lorsqu'un programme est jugé correct syntaxiquement par le programme compilateur, il peut être nécessaire d'utiliser de tels outils pour mettre le programme au point.

Cependant, pour des versions de BASIC interprétées, ces outils sont beaucoup moins utiles, puisqu'il est toujours possible d'insérer des instructions de sortie. On voit ici l'intérêt de numéroter les instructions d'un programme de 10 en 10 par exemple, car cela permet d'insérer et d'effacer des instructions de sorties lors de la mise au point.

Avant de terminer ce paragraphe, il est important de faire remarquer au programmeur qu'il ne doit pas prendre l'habitude de se lancer dans l'écriture d'un programme sans l'avoir bien analysé sur le papier. En effet, et c'est l'un des dangers des systèmes interprétés une mauvaise analyse aboutit toujours à de mauvais programmes.

En effet, si l'on s'aperçoit qu'une première version ne fonctionne pas, on peut bien sûr essayer de la corriger directement, mais, si le nombre de corrections est trop important, il vaut mieux reprendre le programme ou la partie de programme incriminée et la réécrire sur papier pour obtenir une version définitive claire et bien structurée. Cela est important notamment si le programme est destiné à être utilisé ou éventuellement modifié par la suite.

Les principaux types d'erreurs et leur interprétation

Il s'agit d'un problème important qui déroute ou dégoûte plus d'un programmeur débutant lorsqu'il ne comprend pas ses erreurs. Ceci n'est guère facilité par le laconisme des messages d'erreurs et leur inadéquation à préciser ce qui a pu se passer au niveau du langage évolué et non pas au niveau du système. Enfin, bien sûr, la langue utilisée (bien souvent ces messages sont en anglais) peut poser des problèmes de compréhension des messages, voire d'interprétation.

L'objet de cet ouvrage n'est pas de donner une interprétation des messages d'erreurs des différents systèmes, mais de prévenir l'utilisateur des erreurs les plus fréquentes et de la manière de rechercher une erreur.

Les erreurs de syntaxe

Ces erreurs sont relativement faciles à trouver. Il faut d'abord examiner chaque élément de l'instruction : le numéro de

l'instruction (existe-t-il déjà ?), les mots clés sont-ils bien orthographiés ? les noms de variables et les constantes sont-ils corrects ? les séparateurs (blanc, point, virgule, apostrophe...) sont-ils présents et bien placés ?

Ensuite, s'il s'agit d'une instruction complexe, on devra étudier sa structure. Pour les instructions arithmétiques, n'a-t-on pas oublié des parenthèses ou des opérateurs, des séparateurs ? n'y a-t-il pas de mélange avec des variables non numériques ? Pour les instructions de tests, il faut d'abord vérifier l'instruction conditionnelle : les opérateurs relationnels ou logiques sont-ils bien placés ? Ensuite on examinera l'instruction qui suit la clause ALORS (THEN). Pour les instructions FOR, il faut bien sûr vérifier les expressions utilisées pour les valeurs initiales, terminales et le pas, il faut vérifier que l'on a bien une instruction NEXT associée avec le bon paramètre de contrôle.

Enfin, pour les autres instructions, on vérifiera que les étiquettes des instructions GO TO ou GO SUB existent bien, que les séparateurs dans les instructions d'entrées-sorties sont bien placés. Il faut également vérifier que les variables indicées ont été déclarées dimensionnées.

Les erreurs d'exécution

On distinguera deux cas : celui où le programme a déjà marché et le cas contraire :

Si le programme n'a jamais marché, il peut s'agir d'une erreur de logique au niveau de l'analyse ou de l'écriture du programme. S'il ne sort aucun résultat, on devra vérifier que le programme ne "boucle" pas. Si le programme sort des résultats faux, on devra essayer de localiser l'endroit où se produit l'erreur.

S'il s'agit d'un calcul, il faudra vérifier, si les instructions arithmétiques sont correctes, notamment que la hiérarchie des opérateurs est respectée.

Si le programme fait « le contraire » de ce qu'il aurait dû faire, il faudra vérifier que les conditions sont bien définies dans les instructions de tests.

Il faut aussi vérifier qu'il ne s'agit pas simplement d'une erreur au niveau des entrées sorties.

Dans le cas où l'on utilise des boucles FOR, il faut vérifier les bornes et surtout les problèmes d'intervalles, un tour en trop ou en moins dans une boucle peut donner des résultats aberrants.

Si tout cela est correct, il faudra avoir recours à l'insertion d'ordres PRINT pour déterminer de façon précise le groupe d'instructions à partir duquel les résultats sont faux.

Cas où le programme a déjà fonctionné

Dans le cas où le programme a déjà fonctionné, les erreurs que l'on peut retrouver après plusieurs exécutions sont dues essentiellement au fait que le jeu des données varie d'une exécution à l'autre. Deux cas peuvent se présenter. Le premier cas correspond bien souvent à des données particulières qui ont donné le contrôle, à des parties du programme qui n'ont pas encore été utilisées, révélant ainsi des erreurs de programmation non encore détectées. Dans certains cas également, la possibilité de ces données particulières n'a pas été prévue, et cela provoque des erreurs d'exécution : débordement (overflow), division par zéro..., etc. Il est donc important de prévoir des jeux de données tests aussi étendus que possible et de prendre en compte tous les cas particuliers. Le deuxième cas correspond à des fonctionnements aberrants du programme consécutifs à des données erronées ou à des mauvaises manipulations de l'utilisateur. Ce type d'erreurs est bien souvent inattendu par la personne, car il n'a pas envisagé que l'on puisse faire ce type d'erreurs. Néanmoins, ce type d'erreurs teste la robustesse du programme. En effet, quel que soit le problème à résoudre, si le programme est destiné à être utilisé par des personnes autres que celui qui l'a écrit, il est de la plus haute importance de prévoir des « garde-fous », et de prévoir des messages d'erreurs pour toute mauvaise manipulation ou toute donnée erronée. Il se peut d'ailleurs que le corps principal du programme représente un nombre d'instructions qui est inférieur au nombre d'instructions destinées à traiter les erreurs de ce type !

Ce travail est donc assez ingrat, mais il est nécessaire si l'on veut obtenir des programmes robustes à toute épreuve.

Avant de mettre un programme en service, il est donc conseillé de le mettre entre les mains de personnes non expérimentées pour tester cette robustesse. En cette matière, c'est toujours l'utilisateur qui doit primer, et il ne sert à rien de se défendre en disant que celui-ci n'a pas respecté le manuel d'utilisation ou qu'il a fait une mauvaise manœuvre. Tout programme doit être protégé au maximum contre toute manipulation erronée.

Enfin, pour terminer ce paragraphe, il est également important de prévoir des messages d'erreurs suffisamment explicites pour que l'utilisateur comprenne ses erreurs. Ceci est particulièrement important dans tout programme interactif. En particulier l'utilisation de tests du type SI ERREUR ALLER A (ONERR GO TO...) est recommandée pour tout programme destiné à des utilisateurs non prévenus des messages habituels d'un système BASIC.

CONCLUSION

Dans cet ouvrage, nous avons présenté les différentes caractéristiques et possibilités du langage BASIC. Les quatre premiers chapitres concernent l'étude des structures et instructions standard du langage. En même temps, les principes et les techniques de base de la programmation ont été présentés de manière progressive pour permettre au débutant de s'initier plus facilement et rapidement à la programmation.

En effet, selon toutes les statistiques, le langage BASIC est le langage qui permet de réaliser le plus rapidement des programmes opérationnels. Notre expérience d'enseignement confirme également que c'est le langage le plus rapidement assimilé pour des utilisateurs non professionnels de l'informatique (gestionnaires, économistes, médecins, comptables, secrétaires,... etc.). Cela ne veut pas dire que BASIC soit le meilleur langage de programmation : il existe des langages comme PASCAL qui sont bien mieux structurés et donnent des programmes bien plus faciles à lire et à maintenir.

Il n'en reste pas moins que, du fait de son caractère interprété, BASIC est un langage où le système d'exploitation de la machine devient quasi transparent à l'utilisateur débutant.

Pour le professionnel ou l'enseignant, la disponibilité de BASIC sur des micro-ordinateurs autonomes permet également la mise en écriture et la mise au point rapide de programmes pour tester un algorithme, obtenir un résultat rapidement, visualiser une courbe, consulter et mettre à jour des petits fichiers, etc.

Dans les derniers chapitres (5, 6 et 7) ont été présentées des extensions du langage qui ne sont pas standardisées. Nous avons délibérément choisi de présenter les possibilités existantes sur des petites machines. Là aussi, ce qui est important, c'est d'avoir saisi les concepts de base : la notion de primitive à un système de fichiers, la génération de vecteurs, de formes, les changements d'origine et d'échelle en graphique.

Bien sûr, tel ou tel système pourra réaliser ces concepts de manière plus souple que tel autre, et les exemples donnés n'ont pour but que de montrer ce qu'il est possible de faire avec un système minimal.

Contrairement à ce que l'on pense bien souvent, les systèmes micro-ordinateurs dits personnels ne sont pas les moins performants ni les moins souples. Certes, ils manquent quelquefois de « garde-fous », et les systèmes de fichiers y sont bien souvent rudimentaires, mais cela évolue vite grâce à la possibilité d'amortir des logiciels plus sophistiqués sur un grand nombre d'unités.

Nous espérons donc que cet ouvrage a répondu à l'attente de tous les utilisateurs débutants et qu'il leur permettra non seulement de bien maîtriser le langage BASIC, mais de développer leurs propres applications et d'en imaginer de nouvelles plus complexes.

Enfin, ceux pour qui le langage BASIC n'est qu'un début, nous pensons que, malgré ses imperfections, cet apprentissage d'un langage évolué constitue un tremplin très utile pour la maîtrise rapide de tout autre langage de programmation.



APPENDICES

APPENDICE I

RAPPELS DE NUMERATION BINAIRE

La langage binaire

Définition : Le langage binaire est le langage dont l'alphabet est composé de deux caractères :

$$A = \{0, 1\}$$

– Les mots de ce langage sont donc 0, 1, 10, 11, 100, 101, 110, 111...

On peut l'utiliser comme système de codage pur d'un ensemble.

Exemple :

$$0 \leftrightarrow a_1$$
$$1 \leftrightarrow a_2$$
$$10 \leftrightarrow a_3$$

Système de numération binaire

Le langage binaire peut également être utilisé comme *système de numération* en associant à chaque caractère binaire une valeur correspondant à une puissance de 2.

Ainsi, à la position la plus à droite, on associera la puissance 0, à la position suivante en allant vers la gauche la puissance 1, puis la puissance 2, etc. C'est ce que l'on appelle un système de numération par position.

Principe

Un nombre N sera représenté en base b par :

$$N = a_0 \times b^0 + a_1 \times b^1 + a_2 \times b^2 + \ldots a_n \times b^n$$

avec :

$$0 \leqslant ai < b$$

Cas particuliers

– *si b = 10 :*

$$N = a_0 \times 1 + a_1 \times 10 + a_2 10^2 + ... a_n \times 10^n$$

$$0 \leqslant ai < 10$$

a_0 représente le chiffre des unités, a_1 celui des dizaines a_2 celui des centaines...

Les *ai* correspondent à un caractère décimal.

– *Si b = 2 :*

$$N = a_0 \times 1 + a_1 \times 2 + a_2 \times 2^2 + ... a_n \times 2^n$$

$$0 \leqslant ai < 2 \qquad A = \{0,1\}$$

Les *ai* correspondent à un caractère binaire.

Exemples :

– *Le nombre décimal 1789 est interprété comme la valeur :*

$9 \times 1 + 8 \times 10 + 7 \times 10^2 + 1 \times 10^3 = 9 + 80 + 700 + 1\ 000$

– *Le nombre binaire 1 101 est interprété comme :*

$1 \times 1 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 = 1 + 0 + 4 + 8 = 13_{10}$

Ce nombre représente l'entier 13 en décimal.

Remarque. – Le fait de considérer le caractère le plus à droite comme le chiffre de poids le plus faible est arbitraire et, dans la mesure où l'on écrit de gauche à droite, cela peut paraître contre nature. Il faut rappeler que le système de numération par position nous vient des mathématiciens arabes, qui écrivent de droite à gauche.

Lorsque l'on considère le système de numération binaire, un chiffre binaire est appelé *bit* (contraction des mots anglais « Binary digit »). En français on dit aussi *eb* (élément binaire).

Propriété

D'après ce que l'on a vu ci-dessus, un mot binaire de n bits permet de représenter les nombres entiers suivants :

$$0 \leqslant N \leqslant 2^n - 1.$$

Exemple :

– Un mot de huit bits permet de représenter les entiers de 0 à $2^8 - 1 = 255$.

Nombre de bits nécessaires à la représentation d'un nombre entier.

Inversement, si l'on veut représenter un nombre N en binaire, il faudra un nombre de $n = \lceil \text{Log}_2 N \rceil$ bits, où le symbole $\lceil \quad \rceil$ indique l'entier immédiatement supérieur à la valeur calculée.

En pratique, si l'on ne dispose pas de table de logarithme, on peut déterminer ce nombre n en encadrant le nombre N avec des puissances de 2.

$$2^{n-1} \leqslant N \leqslant 2^n$$

Dans ce cas, l'on sait que pour représenter N, il faut n bits.

Exemple :

Si N = 746, on a :

$$n = \lceil \text{Log}_2 N \rceil = 10$$

En effet :

$$2^9 = 512 < 746 < 2^{10} = 1024$$

Conversion binaire décimale

La conversion d'un nombre binaire en un nombre décimal résulte du principe de numération.

L'algorithme de conversion peut se résumer ainsi :

1. Se positionner à droite du nombre ;
2. Initialiser N à 0 et n à 0 ;
3. Lire le caractère suivant vers la gauche ;
4. Si ce caractère est un blanc, alors s'arrêter ;
5. Si ce caractère est un zéro, *alors* faire $n + 1$ et continuer en 3. *Sinon,* calculer $N + 2^n$ et continuer en 3.

Exemple :

_ *1 0 1 1 1 0* représente en décimal le nombre N = 2 + 4 + 8 + 32 = 46

Remarque :

- un nombre pair se termine par un bit 0 ;
- un nombre impair se termine par un bit 1.

Conversion décimale-binaire

Si l'on considère la division par 2 d'un nombre entier N, on a :

$$N = 2 \times q_0 + r_0 \qquad 0 \leqslant r_0 \leqslant 1$$

Si $q_0 > 2$, on a :

$$q_0 = 2\,q_1 + r_1 \qquad 0 \leqslant r_1 \leqslant 1$$

Soit $N = 2^2 q_1 + 2\ r_1 + r_0$
Si $q_1 > 2$:
$q_1 = 2\ q_2 + r_2 \qquad 0 \leqslant r_2 \leqslant 1$
Soit :

$$N = 2^3 q_2 + 2^2 \times r_2 + 2 \times r_1 + r_0$$

Si l'on poursuit ainsi les divisions par 2 des quotients successifs tant que cela est possible on obtient :

$$N = 2^i q_i + 2^{i-1} r_{i-1} + \ldots\ldots + 2^3 r_3 + 2^2 r_2 + 2 \times r_1 + r_0$$

avec $0 \leqslant q^i \leqslant 1$

Ceci montre que les restes des divisions par 2 successives des quotients représentent par définition, les chiffres binaires associés au nombre N.

Exemple : Soit à convertir N = 59. Les divisions successives sont :

59	2				
1	29	2			
	1	14	2		
		0	7	2	
			1	3	2
				1	1

Soit en binaire : 111011 (on relit les restes du bas vers le haut).

Les représentations octale et hexadécimale

En pratique les nombres binaires sont difficiles à manipuler à cause de leur longueur et de leur monotonie (suite de 0 à 1).

Pour condenser ces nombres on utilise des représentations plus compactes.

Représentation octale

Il s'agit du système de numération à base 8, qui a pour alphabet :

$$A = \{ 0, 1, 2, 3, 4, 5, 6, 7 \}$$

Le passage de binaire en octal est immédiat en considérant les groupes de trois bits successifs d'un nombre binaire.

000	0	100	4
001	1	101	5
010	1	110	6
011	3	111	7

Ainsi, le nombre binaire 101110 s'écrit 56 en octal, car 101 = 5, 110 = 6.

La conversion inverse est également immédiate : chaque chiffre octal est remplacé par son équivalent binaire.

Représentation hexadécimale

Il s'agit du système de numération à base 16 avec pour alphabet :

0, 1, 2...9, A, B, C, D, E, F

En décimal	10	A	1010 en binaire
	11	B	1011
	12	C	1100
	13	D	1101
	14	E	1110
	15	F	1111

Le passage de binaire en hexadécimal se fait cette fois en considérant des groupes de quatre bits successifs et en les remplaçant par un chiffre hexadécimal.

Soit, par exemple, le nombre binaire 10111001

Il se traduit en : B9 en hexadécimal, car :

1011 = B
1001 = 9

La conversion en sens inverse est similaire : chaque chiffre hexa correspond à une suite de quatre bits.

Le décimal codé binaire (DCB) :

Certains ordinateurs ou machines à calculer, bien que traitant de façon interne des mots du langage binaire, utilisent pour la représentation des nombres décimaux un *code* qui associe à chaque chiffre décimal sa représentation binaire.

Soit :	0	0
	1	1
	2	10
	3	11
	4	100
	5	101
	6	110
	7	111
	8	1000
	9	1001

Dans ce cas, la représentation des nombres décimaux *n'utilise pas* le principe de la numération binaire au-delà du chiffre 9, mais chaque chiffre d'un nombre est représenté par son code.

Exemples :

- 78 est représenté par : 01111000
- 99 est representé par 10011001

Cela implique que toutes les configurations binaires d'un mot ne sont pas déchiffrables en BCD.

En effet, toutes les configurations de quatre bits allant au-delà de 1001, sont indéchiffrables.

Ainsi, toutes les suites de quatre bits suivantes sont incorrectes : 1010, 1011, 1100, 1101, 1110, 1111.

Addition et multiplication en binaire

1. *Addition :* Le principe de l'addition binaire est similaire à celui de l'addition décimale.

La table d'addition en binaire se limite à quatre combinaisons de digits possibles :

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 10$$

Le seul cas où il y ait une retenue est le dernier. Le digit correspondant au résultat est 0 et la retenue de 1 se reporte sur le digit suivant.

Exemple :

```
   11
   1011        11
 + 1101      + 13
 ------      ----
  11000        24
```

Remarque. – Dans le cas où la taille des mots est fixée à n, le nombre maximum que l'on peut obtenir pour des nombres strictement positifs est $2^n - 1$. Si l'on ajoute deux nombres dont la somme est supérieure à ce $2^n - 1$, on obtient un débordement de capacité.

2. *Multiplication*

Les tables de multiplication en binaire sont limitées à :

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

Le seul cas où l'on ait un résultat non nul est le dernier.

Algorithme de multiplication

1 – Initialiser N = 0.
2 – Initialiser le résultat P = 0.
3 – Lire le caractère suivant vers la gauche du multiplicateur.
4 – Si ce caractère est blanc – stop.
5 – Si ce caractère est 0, faire N + 1 continuer en 3.
6 – Sinon, ajouter le multiplicande suivi de N zéros à P faire N + 1 et continuer en 3.

Exemple :

```
      1011
  ×    101
  --------
      1011
    1011
  --------
    110111
```

Remarque. – La multiplication de deux mots de *n* bits nécessite un mot de 2 *n* bits pour le résultat.

Codage des nombres négatifs

Le problème de la représentation des nombres entiers négatifs suppose un codage du *signe.*

Cela nécessite un bit : on peut par exemple prendre le bit le plus à gauche d'un mot de *n* bits et considérer que sa signification est + si ce bit est 0
– si ce bit est 1.

Dans ce cas, cela suppose que les *n* – 1 bits restants sont utilisés pour représenter la valeur absolue.

Exemple :

Le mot 00000101 représente + 5
et le mot 10000101 représente – 5.

Ce système de codification présente plusieurs inconvénients :
– pour effectuer une opération sur des entiers positifs et négatifs, il faut tester systématiquement le bit de signe ;
– il faut définir une opération de soustraction des valeurs absolues ;
– enfin, il existe une configuration 10000000 qui est interprétée comme un zéro négatif.

Dans ce système, on a donc deux codes pour représenter le 0. Ceci n'est guère satisfaisant.

La représentation en complément à 1.

Cette représentation utilise comme convention de représentation d'un nombre négatif la représentation qui consiste à prendre pour nombre opposé d'un nombre celui qui est obtenu en prenant le complément à 1 de chaque bit.

Le complément à 1 de 0 est 1

Le complément à 1 de 1 est 0

Exemple :

+ 6	0110
− 6	1001

Cette convention utilise également le bit le plus à gauche pour représenter le signe.

L'opération de négation est simple et identique sur tous les bits. L'addition bit à bit d'un nombre et de son opposé donne la configuration 1111, qui est le complément à 1 de 0000.

Dans cette représentation on a donc encore l'inconvénient du double zéro (+ 0 et − 0).

Convention en complément à 2

Si l'on considère la soustraction de deux chiffres décimaux 9 − 3 = 6, et si l'on considère le complément à 10 de 3, il s'agit de 7.

Si l'on ajoute 9 + 7, on obtient 16, c'est-à-dire le chiffre 6 associé à une retenue de 1.

Si l'on ignore cette retenue, la soustraction de 3 à 9 est donc équivalente à l'addition sans retenue de 9 et du complément à 10 de 3, soit 7 (ceci est général et le même principe utilisé en binaire est le complément à 2).

Pour obtenir le complément à 2 d'un nombre binaire, il suffit de prendre le complément à 1 et de rajouter 1 au résultat.

Exemple :

Soit le nombre en binaire : 00001011

Son complément à 1 : 11110100 est obtenu en prenant l'opposé de chacun des bits.

Son complément à 2 : 11110101 est obtenu en rajoutant + 1 au complément à 1.

L'addition d'un nombre et de son opposé donne :

```
 00001011
 11110101
---------
100000000
```

On voit donc que cette fois le nombre obtenu sur huit bits est 00000000, la retenue s'est en effet propagée jusqu'au 9[e] bit, qui n'appartient pas au mot de huit bits.

Cette fois on a donc une représentation qui est telle que :

$$a + (-a) = 0.$$

D'autre part, dans ce mode de représentation, le nombre binaire 10000000 est par définition égal à − 128 : en effet, il s'agit d'un nombre négatif et sa valeur absolue est :

$$10000000, \text{ c'est-à-dire } 2^7 = 128.$$

. La convention complément à deux ne possède donc qu'un seul zéro.

Généralisation

La convention en complément à 2 est actuellement généralisée à tous les ordinateurs.

Si l'on a un mot de *n* bits les nombres que l'on peut représenter en convention complément à 2 sont tels que :

$$-\ 2^{n-1} \leqslant N \leqslant 2^{n-1} - 1$$

EXEMPLE :

Sur une machine travaillant sur des mots de seize bits on a donc :

$$-\ 2^{15} \leqslant N \leqslant 2^{15} - 1, \qquad \text{soit : } -\ 32768 \leqslant N \leqslant 32767$$

La représentation des nombres réels

Lorsque l'on veut pouvoir traiter des nombres fractionnaires, il est nécessaire de définir un nouveau mode de représentation des nombres.

Représentation en virgule fixe

Une première solution serait de considérer une partie entière et une partie décimale séparée par une virgule dont la position serait fixée à priori.

Soit un mot de huit bits ; on peut définir la position de la virgule au milieu du mot, ce qui donnerait quatre bits pour la partie entière et quatre bits pour la partie décimale.

$$N = a_3 \times 2^3 + a_2 \times 2^2 + a_1 \times 2 + a_0$$
$$+ a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + a_{-3} \times 2^{-3} + a_{-4} \times 2^{-4}$$

Dans ce cas, on peut représenter des nombres positifs fractionnaires allant de 0000.0000 à 1111.1111, soit de 0.0 à 15.9375.

On voit donc que le champ de variation est très réduit.

Si, de plus, on considère les nombres positifs et négatifs, l'on ne peut plus aller que de − 8.9375 à + 7.9375.

Le repésentation en virgule fixe est donc très limitée pour le nombre de bits dont l'on dispose.

Représentation en virgule flottante

Une autre solution est de considérer un nombre fractionnaire à l'aide d'une représentation exponentielle (logarithmique).

Soit, par exemple, le nombre :

$$1984.128$$

On peut le représenter sous différentes formes équivalentes :

$$19.84128 \times 10^{2}$$
$$0.1984128 \times 10^{4}$$
$$1984\,128 \times 10^{-3}$$

Ainsi, l'on voit qu'en faisant varier la valeur de la puissance, l'on peut faire déplacer la position de la virgule d'où le nom de virgule flottante.

En binaire, cette représentation correspond à un nombre binaire multiplié par une puissance de 2, soit par exemple :

$$11011.1101$$
$$= 110111101 \times 2^{-4}$$
$$= 0.110111101 \times 2^{5}$$

Représentation normalisée

La représentation normalisée est celle qui est caractérisée par la position de la virgule (du point), telle que le premier chiffre significatif soit à droite de la virgule.

Exemple : 0.110111101×2^{5} est une représentation normalisée.

Définition : On appelle *mantisse* l'ensemble des chiffres significatifs (bits) en représentation normalisée.

L'*exposant* est la puissance de 2 associée à cette représentation normalisée.

Définition : Un nombre flottant est caractérisé par un nombre binaire (positif ou négatif) représentant la mantisse associé à un nombre binaire positif ou négatif représentant l'exposant.

Exemple : On peut par exemple définir un nombre flottant par le schéma suivant :

S	EXPOSANT	S	MANTISSE
	8 bits		24 bits

PUISSANCES DE DEUX

2^n	n	2^{n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 676 950 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125

PUISSANCES DE 16 (EN BASE 10)

16^n	n	16^{-n}	
1	0	0.10000 00000 00000 00000	$\times 10$
16	1	0.62500 00000 00000 00000	$\times 10^{-1}$
256	2	0.39062 50000 00000 00000	$\times 10^{-2}$
4 096	3	0.24414 06250 00000 00000	$\times 10^{-3}$
65 536	4	0.15258 78906 25000 00000	$\times 10^{-4}$
1 048 576	5	0.95367 43164 06250 00000	$\times 10^{-6}$
16 777 216	6	0.59604 64477 53906 25000	$\times 10^{-7}$
268 435 456	7	0.37252 90298 46191 40627	$\times 10^{-8}$
4 294 967 296	8	0.23283 06436 53869 62891	$\times 10^{-9}$
68 719 476 736	9	0.14551 91522 83668 51807	$\times 10^{-10}$
1 099 511 627 776	10	0.90949 47017 72928 23792	$\times 10^{-12}$
17 592 186 044 416	11	0.56843 41886 08080 14870	$\times 10^{-13}$
281 474 976 710 656	12	0.35527 13678 80050 09294	$\times 10^{-14}$
4 503 599 627 370 496	13	0.22204 46049 25031 30808	$\times 10^{-15}$
72 057 594 037 927 936	14	0.13877 78780 78144 56755	$\times 10^{-16}$
1 152 921 504 606 846 976	15	0.86736 17379 88403 54721	$\times 10^{-18}$

PUISSANCES DE 10 (EN BASE 16)

			10^n	n			10^{-n}		
			1	0	1.0000	0000	0000	0000	
			A	1	0.1999	9999	9999	999A	
			64	2	0.28F5	C28F	5C28	F5C3	$\times 16^{-1}$
			3E8	3	0.4189	374B	C6A7	EF9E	$\times 16^{-2}$
			2710	4	0.68DB	8BAC	710C	B296	$\times 16^{-3}$
		1	86A0	5	0.A7C5	AC47	1B47	8423	$\times 16^{-4}$
		F	4240	6	0.10C6	F7A0	B5ED	8D37	$\times 16^{-4}$
		98	9680	7	0.1AD7	F29A	BCAF	4858	$\times 16^{-5}$
		5F5	E100	8	0.2AF3	1DC4	6118	73BF	$\times 16^{-6}$
		3B9A	CA00	9	0.44B8	2FA0	9B5A	52CC	$\times 16^{-7}$
	2	540B	E400	10	0.6DF3	7F67	SEF6	EADF	$\times 16^{-8}$
	17	4876	E800	11	0.AFEB	FF0B	CB24	AAFF	$\times 16^{-9}$
	E8	D4A5	1000	12	0.1197	9981	2DEA	1119	$\times 16^{-9}$
	918	4E72	A000	13	0.1C25	C268	4976	81C2	$\times 16^{-10}$
	5AF3	107A	4000	14	0.2D09	370D	4257	3604	$\times 16^{-11}$
3	8D7E	A4C6	8000	15	0.480E	BE7B	9D58	566D	$\times 16^{-12}$
23	8652	6FC1	0000	16	0.734A	CA5F	6226	F0AE	$\times 16^{-13}$
163	4578	5D8A	0000	17	0.B877	AA32	36A4	B449	$\times 16^{-14}$
DE0	B6B3	A764	0000	18	0.1272	5DD1	D243	ABA1	$\times 16^{-14}$
8AC7	2304	89E8	0000	19	0.1D83	C94F	B6D2	AC35	$\times 16^{-15}$

Représentation des caractères

La représentation des caractères en machine se fait par l'intermédiaire d'un code.

Si l'on veut représenter l'ensemble des lettres de l'alphabet et les chiffres, soit un total de 36 caractères, il nous faut déjà 6 bits puisque :

$$32 < 36 < 64 = 2^6$$

Si l'on rajoute les minuscules et des caractères spéciaux, il faut donc 7 bits.

Historiquement, des codes 6 bits ont été d'abord utilisés, puis le code 7 bits avec bit de parité s'est imposé (code ASCII). D'autre part, certains constructeurs utilisent un code de 8 bits (code EBCDIC).

En pratique, actuellement, les codes utilisés sont des codes 8 bits (ASCII ou EBCDIC).

Les tables de ces codes sont données ci-dessous.

TABLE DES CODES ASCII

CARACTERE OU CONTROLE	ASCII (HEXA DECIMAL)	CARACTERES SPECIAUX ET CHIFFRES	ASCII (HEXA DECIMAL)	CARACTERES MAJUSCULES	ASCII (HEXA-DECIMAL)	CARACTERES MINUSCULES	ASCII (HEXA-DECIMAL)
NUL	00	SP (espace)	20	@	40	\	60
SOH	01	!	21	A	41	a	61
STX	02	"	22	B	42	b	62
ETX	03	#	23	C	43	c	63
EOT	04	$	24	D	44	d	64
ENQ	05	%	25	E	45	e	65
ACK	06	&	26	F	46	f	66
BEL	07	/	27	G	47	g	67
BS	08	(	28	H	48	h	68
HT	09	)	29	I	49	i	69
LF	0A	*	2A	J	4A	j	6A
VT	0B	+	2B	K	4B	k	6B
FF	0C	,	2C	L	4C	l	6C
CR	0D	–	2D	M	4D	m	6D
SO	0E	.	2E	N	4E	n	6E
SI	0F	/	2F	O	4F	o	6F
DLE	10	0	30	P	50	p	70
DCA (X-ON)	11	1	31	Q	51	q	71
DC2 (TAPE)	12	2	32	R	52	r	72
DC3 (X-OFF)	13	3	33	S	53	s	73
DC4 (TAPE)	14	4	34	T	54	t	74
NAK	15	5	35	U	55	u	75
SYN	16	6	36	V	56	v	76
ETB	17	7	37	W	57	w	77
CAN	18	8	38	X	58	x	78
EM	19	9	39	Y	59	y	79
SUB	1A	:	3A	Z	5A	z	7A
ESC	1B	;	3B	[	5B	{	7B
FS	1C	<	3C	\	5C	|	7C
GS	1D	=	3D	]	5D	}(ALT MODE)	7D
RS	1E	>	3E	△(↑)	5E	~	7E
US	1F	?	3F	– (←)	5F	DEL	7F

APPENDICE II

DEFINITIONS SYNTAXIQUES DU LANGAGE BASIC

La syntaxe d'un langage se définit à l'aide d'un « méta langage » qui précise les objets du langage et leurs relations, c'est-à-dire les règles de construction des mots et des phrases du langage. Le métalangage le plus utilisé à cet effet est ce que l'on appelle la forme normale de Backus Naur (BNF).

Les métasymboles utilisés sont les caractères :

: =, qui indiquent une définition ;
le caractère | , qui indique une alternative lorsqu'il y a plusieurs possibilités.

On utilise d'autre part les caractères < > pour indiquer un méta-objet du langage défini par la suite.

Exemple :

< Chiffre > : = 1/2/3/4/5/6/7/8/9/0

Si un méta-objet est optionnel, on l'indique en utilisant des crochets [ ].

Si un méta-objet est répété plusieurs fois, on l'indique en utilisant le caractère *

Exemple :

< nombre > : = < chiffre > *

Le caractère blanc est noté b̸.

On a alors les définitions suivantes :

< Caractère > : = < lettre > | < chiffre > | < caractère spécial >
< Lettre > : = A/B/C/D/E/F/G/H/I/J/K/L/M/N/O/P/Q/R/S/T/U/V/W/X/Y/Z

< chiffre > : = 0/1/2/2/3/4/5/6/7/8/9
< Caractère spécial > : = ƀ/'/"/(/)/:/;/./$/ % / # /!/?/&/*/ + / – / = /// ˆ/ > / <
< Nom de variable > : = < lettre > [< lettre | chiffre > *]
(en pratique, sur la plupart des BASIC, on est limité à deux caractères alphanumériques)
< Nom de variable entière > : = < nom de variable > %
< nom de variable chaîne de caractère > : = < nom de variable > $
< Entier > : = [+ | –] < chiffre > *
< Réel > : = [+ | –] < chiffre > * [. < chiffre > *]
[E [+ | –] < chiffre > *]
< Chaîne de caractères > : = "[< caractère > *]"
< Ligne d'instructions > : = < n° d'instructions > [< instructions : > *] < instruction > ®
(® symbolise le retour à la ligne)
< N° d'instruction > : = < chiffre > [< chiffre > *]
< Instruction > : = < remarque > | < instruction de déclaration > | < instruction exécutable > .
< Remarque > : = REM [< caractère > *]
< Instruction de déclaration > : = < définition de donnée >
< définition de fonction >
< définition de liste ou tableaux >
< Définition de donnée > : = DATA [< donnée > ,]* < donnée >
< Donnée > : = < entier > | < réel > | < Chaîne de caractères >
< Définition de liste ou tableau > : = DIM [< variable >
< dimension >]*
< variable >
< dimension >
< Variable > : = < nom de variable >] < nom de variable entière >] < nom de variable chaîne de caractères >
< Dimension > : = (< variable numérique > [, < variable numérique >]*)
< Variable numérique > : = < entier > | < nom de variable > |
< nom de variable entière >
< Définition de fonction > : = DEF FN < nom de fonction >
= < expression arithmétique >
< Nom de fonction > : = < lettre > [< lettre >]
< Instruction exécutable : = < instruction d'affectation > |
< instruction de test > |
< instruction d'itération > |
< instruction d'entrée/sortie > |

< instruction de branchement > |
< instruction d'aiguillage >
< instruction fin > |
< instruction d'arrêt > |

< Instruction fin > : = END

< Instruction d'arrêt > : = STOP

< Instruction de branchement > : = < mot de branchement >
< n° d'instruction >

< Mot de branchement > : = GO TO/GO SUB

< Instruction d'aiguillage > : = ON < expression arithmé-
tique >
< mot de branchement >
< n° d'instruction >
[< n° d'instruction >]*

< Instruction d'entrée sortie > : = < instruction d'entrée > |
< instruction de sortie >

< Instruction d'entrée > : = < instruction de lecture de donnée
définie dans le programme > |
< instruction d'entrée externe >

< Instruction de lecture de donnée interne > : =
READ < variable > [, < variable >]*

< Instruction d'entrée externe > : =
INPUT [< chaîne de caractères > ;]
< variable > [, < variable >]* |
GET < variable >

< Instruction de sortie > : = PRINT < expression >
[, | ; < expression >]*

< Instruction d'affectation > : = < instruction arithmétique > |
< instruction de manipulation
de chaîne de caractères >

< Instruction arithmétique > : = [LET] < variable > [(< indi-
ce >)] = < expression arithméti-
que >

< Expression arithmétique > : = [+ | −] < terme > |
< expression arithmétique >
< op d'addition > < terme >

< Terme > : = < facteur > | < terme > < opération de multi-
plication > < facteur >

< Op d'addition > : = + | −

< Op de multiplication > : = * | /

< Facteur > : = < variable numérique > | < constante numéri-
que > |
< fonction > | (< expression arithmétique >) |
< expression arithmétique > ↑ < expression
arithmétique >

< Fonction > : = < nom de fonction > (< facteur >)
< Instruction de manipulation de caractères > : =
< Variable chaîne > = < expression chaîne de caractères >
< Expression chaîne de caractères > : = < variable chaîne > |
< expression chaîne >
< opérateur de concaténation >
< variable chaîne >
< Opérateur de concaténation > : = +
< Instruction de test > : = IF < condition > THEN < instruction >
< Condition > : = < expression relationnelle > | < expression logique >
< exp. relat. > : = < expression arithmétique | expression chaîne > < opérateur relationnel >
< expression arithmétique | expression chaîne >
< Opérateur relationnel > : = < | > | > = | < = | = | < >
< Expression logique > : = NOT < expression relationnelle > |
< expression logique > < opérateur logique > < expression logique > |
< expression relationnelle > < opérateur logique > < expression relationnelle >
< Opérateur logique > : = AND | OR
< Instruction d'itération > : = FOR < instruction arithmétique >
TO < expression arithmétique >
STEP < expression arithmétique >
< instructions > * NEXT < variable numérique >

INDEX

BIBLIOGRAPHIE SYBEX

1. LIVRES EN FRANÇAIS

C1 Introduction aux microordinateurs à usage individuel et commercial
C2 Lexique microprocesseurs
C3 Programmation du 6502
C4 Les microprocesseurs
C5 Techniques d'interface aux microprocesseurs
C6 Programmation du 6800
C780 Programmation du Z80
D802 Applications du 6502
PB01 La pratique du Basic
PB02 Introduction au Basic
PA01 Introduction au Pascal

2. FORMATION PERSONNELLE (AVEC CASSETTES)

Chaque cours a été enregistré « live » lors d'un séminaire public, et comprend un livre spécial (sauf SC12 et des cassettes audio.

SC10 Introduction aux microprocesseurs
SC11 Introduction au langage Basic
SC12 Microordinateurs individuels
SC13 Introduction au langage Pascal
SA1 Les microprocesseurs
SA2 Programmation
SB3 Applications militaires
SB5 Tranches de bit
SB6 Applications industrielles
SB7 Techniques d'interface

3. LIVRES ET CASSETTES EN ANGLAIS

Plus de 50 titres. Catalogue détaillé sur simple demande.

4. COMPUTEACHER

Cours d'autoformation complet avec carte microordinateur, cassettes de programme et instructions, manuels.

LES LANGAGES DE PROGRAMMATION CHEZ SYBEX

PB01 *Le Basic par la pratique,* J.-P. Lamoitier
PB02 *Introduction au Basic,* P. Le Beux
PA01 *Introduction au Pascal,* P. Le Beux
PA02 Le Pascal par la pratique (à paraître), P. Le Beux.
P320 *The Pascal Handbook,* J. Tiberghien
B245 *Inside Basic games,* R. Mateosian
P340 *Pascal for Engineers/Scientists* (à paraître)
P350 *Fifty Pascal Programs* (à paraître)

et de nombreux autres titres à paraître.

POUR UN CATALOGUE COMPLET DE NOS PUBLICATIONS

EUROPE
18, rue Planchat
75020 Paris
Tél. : (1) 370.32.75
Télex : 211801

U.S.A.
Sixth Street
Berkeley, CA 94710
Tel. : (415) 848.8233
Telex : 336311



Achevé d'imprimer le 28 février 1981
sur les presses de l'Imprimerie Delmas
à Artigues-près-Bordeaux.

Dépôt légal : 1er trimestre 1981.
N° d'impression : 32073.

INTRODUCTION AU BASIC

Pierre LE BEUX

L'ouvrage

On peut dire que le BASIC n'est pas un langage d'informaticien et que le développement des micro-ordinateurs, rendant l'informatique accessible à toutes les professions, favorise son extension.
La simplicité de la syntaxe et la possibilité de développer de façon directe et immédiate des programmes adaptés aux besoins d'utilisateurs divers : techniciens, personnel de secrétariat, administrateurs, professions libérales... expliquent le succès de ce langage.
Cet ouvrage s'adresse au débutant et ne requiert donc aucune formation préalable aux techniques de l'informatique. Les différents concepts et techniques y sont présentés de façon progressive et pédagogique avec de nombreux exemples de programmes qui ont tous été testés sur des matériels de type micro-ordinateur.
Il constitue donc un ouvrage de référence couvrant tous les aspects du langage actuellement disponibles sur les différents matériels qui vont du micro-ordinateur aux systèmes de temps partagé.

L'auteur

Pierre LE BEUX est ingénieur de l'Ecole centrale et docteur en informatique de l'Université de Californie. Il est coauteur, chez SYBEX, du best-seller mondial *Les Microprocesseurs : Techniques et Applications.*

Autres livres dans cette série :

PB01 : « Le BASIC par la pratique », 50 exercices.
PA01 : « Introduction au PASCAL », du même auteur.

PB02

INTRODUCTION AU BASIC

PIERRE LE BEUX



Document numérisé avec amour par

AMSTRAD CPC

MÉMOIRE ÉCRITE



https://acpc.me/